Software Development - web version of ENCAL

William R Furnass

Computer Science
2002/2003

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)_____

Summary

This final year project describes the development of a new version of ENCAL: a Computer Based Learning tool designed to help teach children about the structure of algebraic expressions.

The process of creating this software involved gaining an understanding of both the existing system and the need for a new object orientated Java version. A software design methodology was chosen to suit the project and the issues involved in working on large development projects were explored. The user interface of the existing version of ENCAL was then recreated using Java's Swing components. The internal structures and methods of the new version required an understanding of recursive data structures; in particular, binary expression trees. These structures had to be designed so that the software would be easily extendable in the future. Finally, the software was tested and evaluated to see if the project objectives had been met.

Acknowledgements

I would like to thank Dr. Vania Dimitrova for supervising this project; her expertise in the area of HCI and software engineering has been very useful, as has her plentiful enthusiasm.

Thankyou, Dr. Andrew Harrop, for being so helpful and approachable, for providing a useful evaluation of my program and for designing such an interesting piece of software; knowing that Encal v2.04 may be used for educational purposes (were I to fully complete the project) spurred me on try my hardest.

I would also like to thank Prof. Thomas Green for his advice and opinions on all matters regarding Encal, programming and computer science, especially on creating and managing tree data structures.

Thankyou, Ken Tait, for the correspondance, the meeting and, most importantly, the example programs, which aided my understanding of recursive methods and data structures immensely.

Thankyou, Dr. Kevin McEvoy, for your understanding, advice and support, and for your help in rescheduling my project.

Thankyou, Dr. Alison Hood, for your diagnosis and your assistance.

Thankyou, Dr. Sarah Fores, for coordinating the final year projects.

Last, but not least, I would like to thank Dr. Eric Atwell for assessing this project.

Contents

Chapter 1

Introduction

ENCAL is a computer-based learning tool for teaching children how to use a calculator and correctly form calculations. It aids in the understanding of the precedence of numerical operators, specifically addition and multiplication and in the development of mental models of calculation. Its effectiveness is largely due to its use of "multiple equivalent linked representations" of calculations [8]. It can be seen from a screen capture of the main interface of ENCAL (fig 1.1) that the three forms of representation are an iconic representation (shown as books on shelves), a calculator and a data tree.

That screenshot was taken from version 2.03 of ENCAL, which was designed by Dr. Andrew Harrop, a member of the Computer Based Learning Unit of the University of Leeds, as part of his PhD research project [8]. Dr. Harrop, now a Post-Doctoral Research Fellow, did not code the program himself; this task was undertaken by Kenneth Tait, who was a Principal Research Fellow for the same unit and who was also Dr. Harrop's supervisor at the time. The software was written using Asymetrix Toolbook II Instructor 5.0, which is a tool that is used amongst other applications for producing educational software. Unfortunately, as Dr Harrop and Kenneth Tait realised when they tried to expand on Encal's initial functionality, Toolbook is simply not powerful enough to cope with a program as demanding as ENCAL. As a result, v2.03 often runs out of memory then crashes and can be slow to act on a given input.

In the summer of 2002, Dr. Vania Dimitrova, a lecturer for the University of Leeds' School Of Computing and a member of the afore-mentioned CBL Unit, proposed a final year project to be undertaken by School of Computing undergraduate, in which a new and hopefully improved version of ENCAL would be created. The new software would be coded in a fully-fledged programming language, a language that is widely known and understood and which, if used carefully, can be used to produce *robust*, *reliable* and *responsive* programs.

The language of choice was Java, which is ideal for avoiding the problems experienced with ENCAL v2.03. Throughout this report, the latest Toolbook version of ENCAL is referred to as *v2.03*; the Java

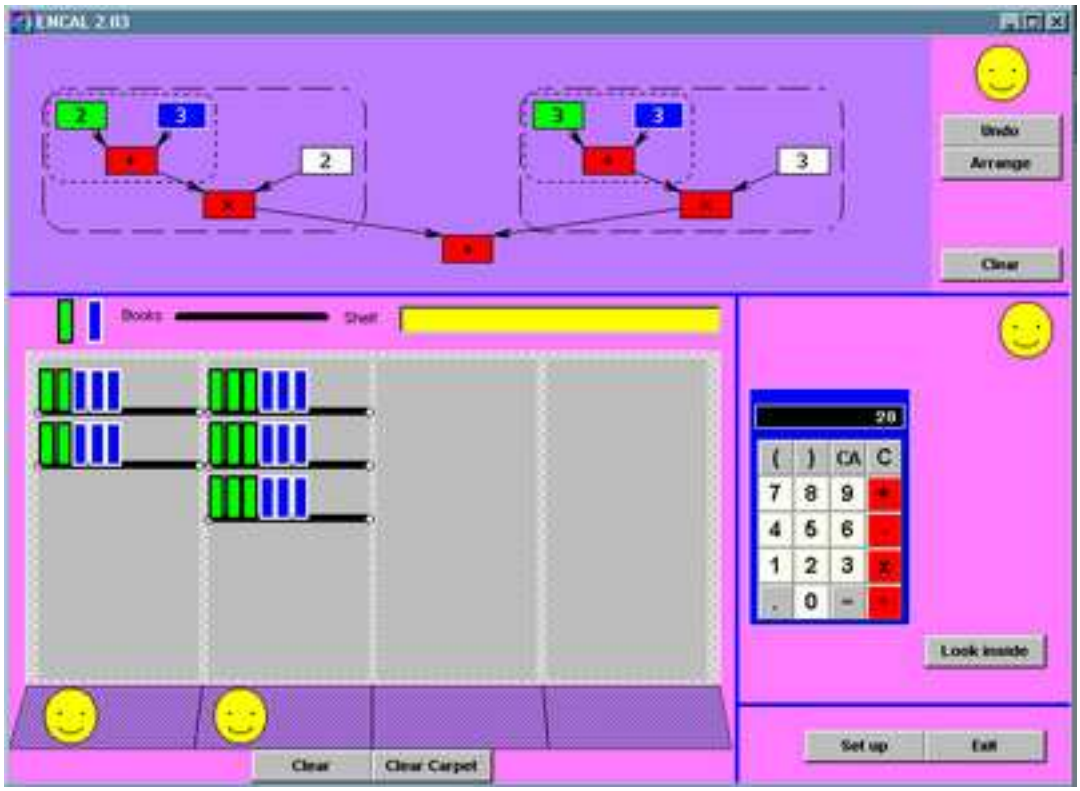

Figure 1.1: A screenshot of an older version of ENCAL (v2.03)

version developed as part of this project is referred to as `v2.04`. Java programs are easily extendable, due to the languages object-orientated nature and great documentation and very portable. A version of the Java Runtime Environment (required to run any Java program) is available for almost all modern computer systems, including Linux, UNIX, Win32, Mackintosh systems and now, even certain mobile phones [16]. Java applets can be embedded in web pages and safely executed on a computer with that virtual machine software installed [4].

This project charts the problems encountered while designing and coding a newer version of ENCAL. The (slightly reworded) **aim** of this final year project was to

> *R*ecode ENCAL from scratch as a Java application, using Swing and the Model-View-Controller (MVC) design pattern. The essential functionality of v2.03 of ENCAL must be preserved in my proposed application; extensions suggested by my client should also be considered.

To elaborate on those terms: Swing components are a subset of the Java Foundation Classes (JVC) "which encompass a group of features to help people build graphical user interfaces (GUIs)...The Swing Components include everything from buttons to split panes to tables" [20].

The Model-View-Controller architecture is described in [5] as separating "application data (contained in the *model*) from graphical presentation components (the *view*) and input-processing logic (the

controller)". Using this pattern, a model can be displayed in different ways using several views and can be altered using many controllers. ENCAL maintains infomation on the current expression being manipulated; this could be seen as being a model, while there are three views, each of which displays a different representation of that expression. Also, each view has its own embedded controls, therefore the model-view-controller pattern seemed appropriate for this project.

Project objectives

Each final year project must have a set of objectives that must be addressed in order that the project aim be met. The objectives that were decided upon in the mid-project report are included as follows:

1. Understanding the problem - familiarisation with current version of ENCAL and identification of the extent to which existing features of ENCAL v2.03 can be changed and new features added. Several parts of the original program were implemented in a slightly different way to make the goals of the project slightly easier to reach and to suit the programming language used.

2. The selection of an appropriate software design methodology. This could be deciding to follow a particular existing methodology or composing a methodology to meet the specific needs of this project.

3. The decomposition the problem into independant tasks, such as the design of the GUI and the internal models/data structures, then, because of the use of an object-oriented programming language, the formation of the object model for the system using design tools such as UML. The latter is an important part of Object Oriented Design; it helps determine what classes are need, how they will interact and how object-orientation's key component technologies, inheritance and polymorphism will be used [4].

4. The development of each component independantly and the testing of the main classes using *white box* testing. With this form of testing, the tester can see inside the box i.e. see the code and can then test, for example, each `if` statements and `for` loop in a particular module.

5. The integratation of the modules together in the final program and testing using *black box* techniques. With black box testing, the tester cannot see the code; he/she uses test cases (often based on use cases) to check that provided with a certain input, the program produces the right output. This often plays a part in evaluating a software project's success.

6. Composing the project report with a description of the work undertaken and a short guide of how to use the system. The report will include an evaluation of both the product and process, whether I have met my objectives and how satisfied the client is with my software.

Minimum Requirements

Associated with the preceding objectives were a set of 'minimum requirements', also included in the mid-project report. These were:

1. A detailed specification of the functionality of ENCAL to be developed in the new version.
2. A justified choice of and appropriate methodology for software design and development.
3. Analysis of the problem with justified decisions what data structures to be used for the internal model and how the interface will be designed.
4. Decomposition of the problem in independant tasks, design of appropriate classes and description of the system's object model in a UML format.
5. Development of a system prototype which complies with the basic functionality of ENCAL in terma of statically linking the three external representations. Not all ways of interacting with ENCAL's views may be implemented, but the essential interactions should be possible.
6. Appropriate testing.

With regards to the the fourth requirement, the representations (iconic, calculator and data tree) should all be at least *statically linked*. By that, it is meant that the user will press a button within a representation view to signal to the program that he/she wishes the changes made to that representation be propagated to the other views, then they will be updated. The alternative is *dynamic linking*, whereby any change the user makes is automatically propagated to the other representations. There are, of course, exceptions: propagation of an expression to a representation will only take place if the change(s) made by the user to another representation have a semantic meaning in the former's context. This will be discussed at a later stage.

Project enhancements

Project enhancements are tasks whose scope lay outside that of the minimum requirements, yet were still relevant to this project and of interest to the client. These were:

1. The provision of more advanced operations for manipulating the representations and provide a robust control of the correctness of each representation, as children will probably try to manipulate the representations in many illogical ways.
2. The inclusion of the grouping rectangles around each operand-operator-operand triple in the data tree representation, a feature that is present in v2.03. The client considers these rectangles to be a useful guide in helping to decide whether parentheses have been entered in the correct places and to aide understanding of the order in which the expression has been calculated.

3. The dynamic linking of the representations.

4. The client was not entirely happy with the data tree representation. The v2.03 tree nodes were either perands or operators, but he considered that not to be adequate, as it did not reflect the information displayed when the 'look inside' button was pressed. The 'look inside' box displays a list of operand-operator-operand-result calculations which show the order in which the current expression was evaluated. What was suggested was that the 'local result' be shown within each grouping rectangle.

5. Another possible extension was to present the new version of ENCAL to its intended end users (children aged twelve to thirteen [8]) and examine how it compares to v2.03 to see whether the changes made to the basic functionality have added to or detracted from its usefulness.

6. Dr. Harrop suggested that the ease which the data tree representation could be edited could be improved by having the user click on an operator to place parentheses around it and its two operands [9].

7. Also suggested was the idea of writing the new version of ENCAL as an applet, as opposed to a application. This would result in a program that could be embedded in a web page and used by anybody with an Internet connection and the Java Virtual Machine installed on their system.

Key Achievements and associated modules

On several occasions throughout this project, I overcame a large and sigificant problem; some of these I solved using what I have learnt during my time at the University of Leeds, others required me to think, research and experiment.

The first of these achievements was familiarizing myself with software design methodologies (chapter 4). These were covered to an extent in a module I undertook in my second year, Software Project Management, but the emphasis there was placed on building software as a team and very few specific methodologies were explored, so I couldn't use that much of what I had learnt from that module. Other relevant modules were Introduction to Programming 2, as it covered the Extreme Programming methodology and Object-Orientated Programming, as it contained material on UML use cases and class diagrams (which are both used by many existing methodologies as part of their notation).

The second major problem that was overcome was how to cope with programming a project of such proportions (chapter 5), as I had very limited experience with working on large programs prior to the start of this project. I had to learn how to use the Javadoc program documentation tool and divide my program up into packages, which was taught in the Object-Oriented Programming module. As part of the Software Project Management module, I had to produce a large piece of software, but it was as part of a group and was not particularily successful; as a result, I learnt how not to tackle such a project.

My fourth achievement was the design of ENCAL's core (chapters 7 and 8) which required knowledge of recursive data structures (in particular, binary trees) and recursive operations that could be used

to examine and alter such structures. I was introduced to both of these in my first year as part of the Introduction to Algorithms and Data Structures module. In the second year, the Functional Programming module took this a step further, then in the third year, the Compiler Design module showed how expression trees could be used as part of a object-orientated program, which proved to be particularly relevant and helpful. However, this project has gone beyond what was covered in those modules. It has provided an opportunity to link the theoretical, such as algorithms for converting between infix and postfix notation, with the practical, such as designing and implementing a graphical interface using Swing.

I also found the dynamic linking to be an interesting problem; this was unfortunately not something that had been specifically covered in any module, but using object-orientated techniques and patterns like those that had been taught in Object-Orientated Design, I designed and implemented a relatively sound linking mechanism (see chapter 8).

Chapter 2

Analysis of ENCAL v2.03

ENCAL was designed to help twelve to thirteen year old children learn how to use calculators to solve mathematical word problems. It does this by facilitating the construction and use of mental models which then aid the formation of algebraic expressions. ENCAL allows the user to view an expression as he/she is constructing it, in several different representations which are all semantically equivalent but vastly different in the way they present the information. This helps the child comprehend the connection between the concrete word problem and the abstract expression that is entered into the calculator.

In Dr Harrop's own words, the ENCAL project originally had three aims [7]:

- *"The first aim is to provide a computer-based learning environment which supports the transition from novice conceptual mental models to expert conceptual mental models by utilising an intermediary data-flow external representation between the concrete (i.e. real-world icons), and the abstract (i.e. algebraic symbols) external representations.*
- *The second aim is to incorporate direct manipulation within and between the real-world, data-flow, and abstract representations to support the transition from novice to expert thinking.*
- *The third aim is to provide users with a four-function calculator mental model (to reduce errors brought about by the left to right bias in children) by creating links between the behaviour (i.e. mental model) of a four-function calculator external representation, and the literal, data-flow, and abstract external representations used in problem solving."*

These aims were all fully realized in the Toolbook system, which meant that there was a need to try and incorporate all ideas that comprise these aims into the new version of ENCAL that was built. The first step of this was the analysis of the ENCAL v2.03 system.

Overview of ENCAL v2.03

V2.03 can display the current expression being shown using up to three structurally different representations: one view illustrates the expression in terms of books on shelves in a bookcase, another shows all operators and operands as nodes in a data-tree-like formation, while the third view provides the user with an interface which is very similar to a standard calculator. Throughout Dr Harrop's thesis, these representations have each been referred to using two appellations: iconic/concrete, datatree/intermediate and calculator/abstract, respectively. It is the way that these representations each contain both unique and redundant information on the current expression that make the aforementioned connection between the concrete and abstract easier to visualise.

The user can, if desired, view all three of the expressions simultaneously and alter the expression in any of the views at any stage and it is nearly always the case that a change to one representation will be immediately reflected in all of the other representations that are on display. For these reasons, Dr Harrop has termed these representations *MERLs* (an acronym for *Multiple Equivalent Linked Representations* [8].

I have already briefly discussed the first two aims, but the third is also worth elaborating on. Many children, when trying to evaluate an expression such as $4+3\times5$ will not realise or take into consideration operator precedence, that is, the result of the multiplication should be computed first, then 4 should be added onto the result. Instead children simply work from left to right, evaluating each operand-operator-operand triple as they encounter it, which is the afore-mentioned left to right bias. ENCAL helps the user see exactly how the expression that they have entered into program has been formed and how it will be evaluated, so they can see which parts of their expression will be calculated first. ENCAL can process an expression using one of two methods of evaluation: *left-to-right* and *BODMAS*. BODMAS (Brackets, Order, Division, Multiplication, Addition, Subtraction) is used by traditional scientific calculators and takes operator precedence into consideration. The inclusion of both evaluation methods in the system lets the user easily compare and contrast the two methods and come to a better understanding of how certain operators take priority over others.

Expression propagation and well-formedness

Not all expressions that can be generated using a particular representation can be propagated to the other views. For example, if the expression $2+\times3$ is entered using the calculator keypad, then this has no meaning in either of the other representations. Such an expression is *not-well-formed*. An expression that does have semantic meaning in at least one other representation (e.g. if $4+(2*5)$ was input using the calculator keypad) is *well-formed*.

Generally speaking, when manipulating an expression in a view, a modification will always result in the expression in that representation being either well-formed, not-well-formed or empty. Whichever of the three possible states the representation is currently in is then related back to the user via a smiley face, un-smiley face or complete absence of a face. The client made clear that the face must never look

unhappy, as this may discourage to the child.

If a representation is altered so that it is left in a well-formed state, then the new expression in that view is propagated to the other two views so that all three representations are equivalent.

If, however, a representation is altered in such a way that it is left not-well- formed, then the new expression is accepted and shown by that view, but the other two views are made 'blank' e.g. the abstract view would not display any books or shelves. This is done to prevent the other two representations from showing contradictory information, as there is no guarantee that the change that was made by the user could be propagated.

The third case is that a view is altered so that it is left in the 'empty' state: either the user pressed the view's clear button or he/she cleared the view manually(e.g. he/she removed all books and shelves). All views are capable of supporting the 'empty expression', therefore all representations are then made empty.

The calculator representation

The calculator representation has four distinct visual parts:

The first is the calculator itself. This has a small display window and a keypad upon which there are the following keys:

- The digits zero to nine.
- The four operations supported by ENCAL: addition, subtraction, multiplication and division.
- Left and right parentheses.
- A button, marked 'C', that un-does the last *expression-augmenting change* (the last addition made to the current expression).
- A button, marked 'CA', that clears the whole expression.
- A button, marked '=', that displays the numerical value of the current expression in the display window, if that expression is well-formed.
- A decimal point button, which does not do anything. Dr Harrop considered that allowing the user to input real numbers would over-complicate the system, so this button does not do anything in v2.04 (th Java version) either.

As well as the calculator, there is also a *look-inside* panel, which, if the current expression is well-formed, displays the details of how the expression was evaluated as a series of *operand-operator-operand-result* quadruples. Beneath this panel, there is a button which changes its visibility.

Thirdly, there is a box which displays the current expression in infix form and finally, there is a yellow face to indicate the representation's well-formedness.

Whenever the current expression is changed, either using the keypad or by another representation, the expression box, the look-inside panel and the yellow face are all updated if necessary. If another representation is changed to leave an expression not-well-formed, then it is not propagated and the expression and panel are then cleared and the calculator display window is set to zero.

It is important to note that if two digit buttons are pressed in succession, then they are amalgamated into one number. For example, if 5 is pressed, then 2 is pressed, then they must be combined to get 52 rather than be treated as if they are two separate numbers in a not-well-formed expression.

The iconic representation

The iconic representation uses the metaphor of a bookcase to represent expressions: within the bookcase, there are four bookshelf *columns*, each of which can contain zero to six shelves. Each shelf can contain zero to nine books. To further complicate matters, each book can be either blue or green.

An expression can only be formed from the bookcase if it is in a well-formed state, which is only true if every column is well-formed. A column is well-formed if the number of blue books is the same for every shelf in the column and the number of green books is the same for every shelf also.

If well-formed, then an expression is created by adding together the sub-expressions for each non-empty column. If more than two columns are non-empty, then the left-most pair's sub-expressions are added first. The sub-expression that corresponds to a non-empty column is found as follows: If there is more than one shelf in a column, then the sub-expression is the number of shelves multiplied by the sub-expression for a shelf. This multiplication can be performed in two ways: *shelves first*, in which the number of shelves is the multiplication operator's left operand and *books first*, in which the order of the operands is the other way round. If there is only one shelf then the column sub-expression is just the sub-expression for a shelf. Finally, a sub-expression for a shelf is the number of blue books added to the number of green books if both types exist on the shelf, or the number of books if there is only books of one type on the shelf or 0 if there are no books on the shelf.

The columns are drawn adjacently on the representation's display panel. Beneath each one is a strip of *carpet*, which is where each column's yellow face is displayed.

A book or shelf can be added to the display by dragging on the appropriate icon above the bookshelves, then releasing the mouse button when the cursor is in the right place. A book cannot be added to an empty column, it must be placed on a shelf. When the user tries to add a shelf to a column, all shelves already in that column are shifted up (if necessary) so that all empty space is at the bottom of the column, then the new shelf in the highest empty shelf slot, if there is room. There is a similar method for adding books: all existing books are shifted along so that all empty space is at the right-most end of a shelf, then the new book is added in the left-most empty slot, if there is space.

A book that is already in the bookcase can be moved by dragging it onto a non-empty shelf. It is then added using the procedure described above.

A book can be removed from the bookcase by dragging it down onto the carpet. The carpet can then be cleared at a later date by clicking the 'Clear Carpet' button or the user could decide to return a book

on the carpet to a shelf within the bookcase. The client considered the carpet to be unnecessary and slightly cumbersome to use, so it was agreed that books in ENCAL v2.04 should be removed by simply right-clicking on them.

A shelf can be removed by clicking on a *nail* at either end. If the shelf is empty, then it is removed straight away; if not, a box appears which prompts the user for confirmation as to whether he/she wishes to remove that shelf and all its contents.

Other features of the representation include the error message box, which informs the user when another representation tries to update the bookcase with an expression that involves putting too many books on a shelf or too many shelves in a column. The error message box also infoms the user when another expression tries to update the bookcase with an expression that cannot be shown because it does not comply with the rules on extracting an expression from the bookcase. For example, no expression with a divide operator in can be displayed. Whenever an error message is shown, the bookcase is also cleared as well.

Finally, there is a 'Clear' button which also clears the bookcases and as a result, clears any expression relating to the bookcase as well.

The Data Tree Representation

This representation treats the current expression as a combination of tree *nodes* and *edges* (or *arcs*) between those nodes. For this representation to be well-formed, all nodes must form part of a proper *binary tree*. This means that there should only be one node with no *parent* node and each node should have either zero or two children. Also, every *leaf* node (leaf nodes have no children) should be an integer and every other node should be an operator. If the user breaks any of these rules through altering the representation, then all other representations will be cleared.

Visually, the representation consists of a main panel on which all nodes are displayed and a side bar. The side bar has a yellow face to indicate well-formedness and also has three buttons. There is a button to undo the last change that was made by the user to the representation, a button to automatically arrange all nodes and a button to clear the representation. The client did not consider it essential that the undo button worked in ENCAL v2.04, he said it was merely desirable. The arrange button only auto-arranges the nodes if the representation is well-formed. Using that function, each node is placed at a vertical height that is proportional to its *depth* (distance from the root) in the tree and, if it is an internal node, its children are placed an equal distance horizontally to either side of the node. The root node is placed horizontally in the centre.

Nodes are added by clicking the left mouse button within the main panel. A blank node then appears. Blank nodes have no semantic value, but there can be no blank nodes if the data tree representation is well-formed. A node cannot be added on top of another node.

Right-click on a blank node and a pop-up menu appears. From this menu, the user can assign the node an integer value (using a pop-up input box) or turn it into an operator node. The user can also choose to 'pick' the node; this means that wherever the user next clicks inside the main panel is where

that node will be moved to. Picking and moving only works if the node in question is disconnected. The client expressed a desire to see simpler 'drag and drop' moving in any future version of ENCAL as it will be much quicker and easier. There is also an option to delete the node or cancel the menu.

Nodes can be connected by dragging using the left mouse button from the node the user wishes to be the child, then releasing over the node that is to be its parent. If the parent already has two children, then no edge is added to the tree.

Once a node has been assigned a semantic value, then a pop-up menu still appears when it is right clicked, but the options are slightly different, depending on the type of node. If if is a number node, then its numerical value can be changed, but it cannot be made into an operator node. It can also be *disconnected* from any adjacent nodes. As before, it can be deleted, picked for moving or the user can cancel the menu. A node can only be deleted if it is disconnected. There are several other options, but they are redundant for number nodes.

The operator node pop-up menu allows the user to change the type of operator, but not the type of node. There is also the option to place a bounding rectangle around the node and its children, as discussed in chapter 1. The user can also choose to swap the left and right children or to remove the left and right children. In the case of the latter, the edges connecting the node to that child, the child itself and the child's subtree are all removed from the representation. Again, there are options to delete, disconnect or pick the node and to cancel.

The nodes are coloured so that they correspond to the iconic representation. Operator nodes are always red, nodes that correspond to a number of books of a particular colour are that colour and all other nodes are white.

The setup screen and other points

The three representations are all displayed simultaneously on the main screen. However, this is not the screen that the user sees when the program is started. The *setup screen* that first appears allows the user to change the following:

- The representation that is currently being hidden, as the user can choose for a particular representation to not be displayed or for all to be shown.
- There is also a similar option which allows the user to shade one or none of the representations and choose the level of shading. The clients were warned the that this feature could be difficult to implement using Java; the reply was that it was not important and could be left out of future versions, so it is was not included in ENCAL v2.04.
- The user can choose whether the calculator representation uses left-to-right evaluation or BODMAS evaluation.
- There is an option to select whether the iconic representation creates expressions using *books first* or *shelves first*.

- The visibility of the look-inside panel can be set.
- The visibility of the full infix expression can also be set.

From within both screens, there is the option of returning to the other or exiting the program.

Finally, the main screen is a fixed width and height, so v2.03 does not scale well when run using certain screen resolutions. The client said that he would prefer it if ENCAL's main screen could be scaled to suit the user's resolution.

Chapter 3

Problems with ENCAL v2.03

When I first met with the client and my project supervisor, the current version of ENCAL at the time, v2.03, was demonstrated to me and the problems with it were stressed. Nearly all of these problems were connected with the implementation of the program, rather than the design. All design issues throughout this project were solved by Dr. Harrop and Prof. Thomas Green, a Visiting Professor to the School of Computing. They were both at hand to question on, say, what a particular feature should do. It is for this reason that I did not research Computer-Based Learning apart from reading Dr. Harrop's thesis and papers, as this project was, from the start, solely concerned with software development. It is outside the scope of this project to review CBL technology and to justify the educational principles ENCAL is based upon.

The following section looks at the reasons why the project was proposed and a new version of ENCAL was needed.

3.1 Slow interpretation speed

The first problem a user will notice when working with ENCAL v2.03 is the occasional delay between user input and the resulting change(s) made to the user interface. This is particularily apparent when a change made by the user to one representation, for example, the calculator, causes both other representations to be updated to display that information in a different form. This is not considered by the client to be a major problem with v2.03, but it certainly detracts from the program's usability. Instead, teh project followed the clients' requirements as specified by Dr. Harrop and Prof. Green.

This problem was due, not to the quality of the v2.03 code, but to the limitations of the scripting language used. The Asymetrix Toolbook II Instructor 5.0 system was chosen for the development of ENCAL way before the functionality that the system would have to possess had been properly considered. Toolbook was designed as a tool for writing simple multimedia programs and is mainly comprised

of a set of screen objects and an extended version of the OpenScript scripting language, which is interpretted [18]. According to Kenneth Tait, the reason that v2.03 was often so slow was because of the crude way that OpenScript handled objects; with OpenScript ”there are many occasions in which other types are converted to strings and back to the original type almost as a side-effect of some particular coding syntax. Thus script execution is slow and often inefficient” [18]. This is particular apparent with v2.03 as it is a fairly computationaly intensive application.

3.2 Memory leaks

Another related problem is that of memory leaks. Frequently, after the user has been interacting with the system for a while, an interaction will cause a box to appear containing the Toolbook error message ”Execution suspended: Not enough memory. Close other applications or save this book and try again”. The user can click ”Cancel” to get rid of the message, but when returned to ENCAL, he/she will may find that the system has not fully completed the last operation requested by the user. The client is unhappy with this behaviour as it again detracts from the usability and a message is presented which has no real meaning to the intended users.

Again, it was due to the inadequacies of OpenScript, as Kenneth Tait explains: “The size of this memory or how it can be increased is not known, but what was discovered was that the destruction of screen objects did not free up all the memory associated with the object” [18]. As a result, the program works properly until, eventually, all the working memory is full and the afore-mentioned message is displayed. This kind of situation is referred to as a *memory leak*, which is when memory that is no longer being used by the program is not returned to the system (which in this case will be the Toolbook virtual machine).

When the memory leak was discovered, Kenneth Tait wrote a simple object management system into ENCAL so that “screen objects rather than being destroyed were merely hidden and added to a list of re-usable objects and the next time an object of that type was required it was re-configured, reposition and revealed” [18]. Unfortunately, as ENCAL grew, this ceased to function well enough.

One of the main advantages of rewriting the software in Java is that the Java Virtual Machine has built-in garbage collection: a garbage collector runs in the background and reclaims system memory when there are no more references to that memory [6]. This means that, if a screen object is longer used, then the memory that is used to store it is reclaimed by the garbage collector for the system.

3.3 Lack of portability

Another problem with implementing the system in Toolbook was that it was not particularily portable. As programs written using Toolbook are interpretted using the Toolbook virtual machine, all those who wish to run ENCAL v2.03 on their own computer must have Toolbook installed as well. The other option is to use Toolbook to convert a program written in its own format into HTML, with images stored as

GIFs and widgets (GUI components) as Java applets. However, this process is similar to converting a Microsoft Word document to HTML as the resulting code is often full of bugs and errors and it can be difficult for anybody to then examine the code and augment or alter it in any way.

A Java version of ENCAL would mean that anyone with Java Virtual Machine installed on their system would be able to run it. This would be an improvement over v2.03 as many computers are now sold with a Java Virtual Machine already installed and the JVM is both popular and free to install. I have created a Java application version of ENCAL, but it is possible to convert this manually to an applet, which would make it even more portable. This is a much 'safer' procedure than the automatic applet creation perfomed by Toolbook as the source code would not be computer generated and therefore much easier to maintain and update.

3.4 Not easily extendable

ENCAL v2.03 was not particularily extendable as the OpenScript scripting language that it was written in is not one of most common languages used to create programs. Java, on the other hand, is widely known and understood; much literature and many websites are devoted to it. Also, if a Java program is well designed, then the language's object-orientated nature makes understanding the architecture of the program fairly simple, especially if a tool such as Javadoc is used.

According to the Sun website, "Javadoc is a tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields" [10]. What makes Javadoc documentation even more appealing to the programmer is that the Java API (Application Programmer's Interface) documentation was also created using Javadoc, so any documentation that is created for a particular Java program is in the same style and therefore very easy to understand.

The extendability of any version of ENCAL post-v2.03 was seen by the client to be of considerable importance because he was not happy with the methods with which the data tree was edited. One idea was for me to improve that functionality, as stated in the project enhancements (see Chapter 1). The other was for me to design a system that could easily be understood and augmented by another programmer so that, once I had finished my project, a completely new version of the data tree representation could be simply slotted into my version, as several new data tree designs have already been explored by Dr. Harrop and Prof. Green. Unfortunately, I did not have the time to implement the former, but the latter should now be possible.

Chapter 4

Design methodologies

The first step when setting out to design an build a large piece of software must always be the selection of an appropriate design methodology, as it allows the developer to split the project into stages if necessary, manage user needs and generally organize the project.

General software design processes: the waterfall model and spiral model

Although there are many software design methodologies in widespread use today, they almost all have very similar features. For example, an analysis stage features in most methodologies, but how many times it is passed through and whether it overlaps with the design stage is highly methodology-dependant.

The common ancestor of most modern methodologies is what is known as the *waterfall model*, as discussed in many texts on the subject [12, 14, 21]. The waterfall in its simplest incarnation has very few stages (see fig 4.1, from [14]), the names and definitions of which vary.

What remains constant, however, is the underlying principle of the model, which is that the developer or team of developers should aim to complete each stage before the next stage is attempted.

There are several problems with this approach, the most important of which is that "requirements change over time, as businesses and the environment in which they operate change rapidly" [14]. A design could arrise out of one set of requirements, then, if the requirements change, there is no way of returning to the design stage to take the new requirements into consideration. Fortunately, because I set my own minimum requirements and project objectives, the client could have completely changed his mind about a particular feature and I was not then obliged to re-implement it. Therefore, *requirements drift*, as it is often called, did not really affect my project.

Secondly, "the waterfall model makes no allowance for prototyping and implies that you can get the requirements right by simply writing them down and reviewing them" [12]. Prototyping played an

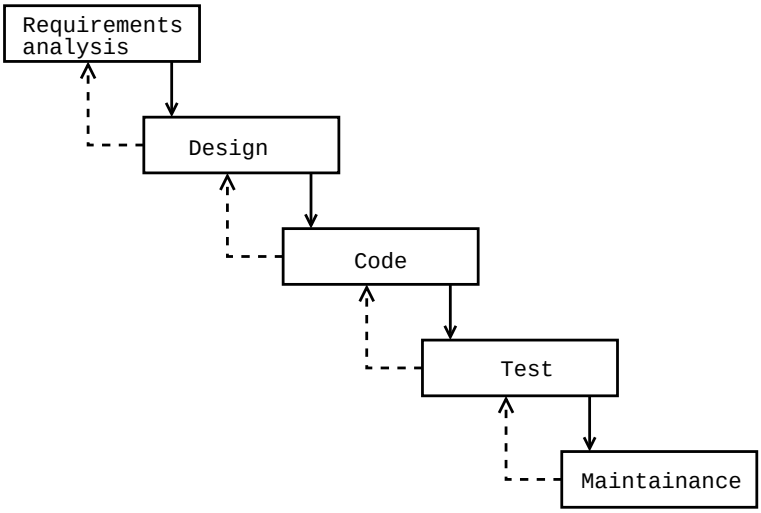

Figure 4.1: The waterfall lifecycle model of software development, adapted from [14]

important part in this project, as the client is not a programmer, so the only way he could see project progress was through prototyping. More importantly, prior to this project, I had had very little experience with working on large software projects and even less experience with Java, so designing a large program all at once, only knowing the very basics about the language's capabilities and limitations would have been rather naive.

Thirdly, one of the nicest features of object orientated programming is that "once you have identified an object, you can often complete the design, implementation and testing of that object seperately from the other parts of the system" [21]. My limited experience with Java meant that unit testing was of considerable importance.

What was needed was a methodology that followed the outline of the waterfall model and yet would allow for the iterative development of an object-oriented program.

First suggested by Barry Boehm in 1998 [14], the spiral model of software development was loosely based on the waterfall model but allowed for both risk analysis and prototyping (see fig 4.2 [14]). However, risk analysis was not really appropriate for this project as so little was known about Java and large programming projects prior to its start. Also, risk analysis is much less important when just producing a new version of a piece of software.

Specific methodologies

There are many methodologies in existance that incorporate the above models as part of larger and more complex processes which were designed to enable teams to manage moderate to large development projects, most of which involve object-oriented programming. The more famous of these include:

- The Catalysis Process
- The Booch methodology

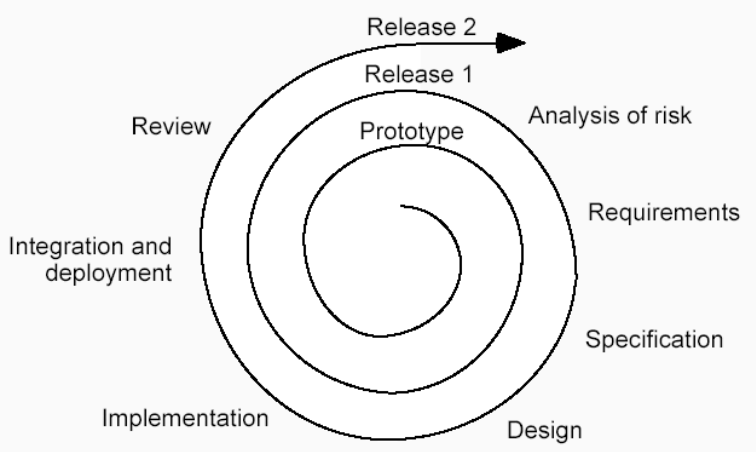

Figure 4.2: The spiral lifecycle model of software development

- The Coad-Yourdon Process
- The Rumbaugh methodology
- The Jacobson methodology
- The (Rational) Unified Process

With very little experience of software design methodologies, I began to research the Unified Process, as it inherits much from earlier methodologies (its authors were Booch, Jacobson and Rumbaugh). After some investigation, it was decided that it was too 'heavy' an approach. The Unified Process was designed to be used to coordinate a large team working over a moderately long time period on a large project [11]. The development of a new version of ENCAL, however, was a project that will be undertaken by just one programmer and the whole process from requirements capture/analysis to evaluation took only a few months, so a much lighter methodology was needed. Also, the volume of documentation required by the process and the amount of reading necessary to understand how to implement the process involved too much work, considering how short the time scale of the project was.

It was suggested by the School of Computing's Nick Efford that a look at the so-called *agile methodologies* might prove fruitful, but again, they were simply too heavy for a project of this size.

According to [2], "Extreme Programming is the most well known of the agile methodologies". It was not suitable for this project for several reasons: the first was that it states that programming must be done in pairs so that mistakes can easily be spotted and debugging can be made easier [3]. The second was because it states that only methods that are necessary should be implemented. One of the requirements for this project was that it should be easily extendable. Therefore, some methods and sections of code may not be particularily useful now, but they will probably be used in the future if/when the version of ENCAL developed for this project is augmented.

The methodology chosen for the development of ENCAL v2.04

As most heavy-weight and light-weight methodologies seemed to be surplus to requirements, the desision was made to fashion a much simpler methodology from the general models. The *phased model*, from [12], lies conceptually between the waterfall and spiral models. It consists of

- an initial requirements gathering and definition stage
- a specification stage
- a series of iterations, each of which involves design, then implementation, then integration.

This model was the basis for the methodology used for the creation of a new version of ENCAL. As the project involved duplicating almost exactly the functionality of an existing piece of software in a 'black box' fashion, there was not much need for extensive requirements and specification documentation. Instead, there was a single stage in which the functionality of ENCAL v2.03 was described in detail and the few changes that would be made to the system were also listed.

There were then several of the afore-mentioned iterations, one for each of the following features:

1. the basic GUI
2. the internal 'core' structures
3. the calculator representation and its integration of the calculator representation with the core
4. the data tree representation and its integration of the calculator representation with the core
5. the iconic representation and its integration of the calculator representation with the core

Note that before the core structures had been investigated, only the first two phases were planned for, as there was no way of knowing how the core was to be broken up.

This methodology allowed me to get to grips again with Java gently, as the design, implementation and testing of the basic GUI helped remind me of the Java syntax and API as well as the basic principles of object-orientated programming, such as composition, inheritance and polymorphism.

The original schedule for the project (See Appendix B) was written before any software development methodology had been considered and was included in my mid-project report. Each stage in the schedule mapped to a particular project objective in an attempt to ensure that all of the latter were met. As a result, it then resembled a simplified waterfall model, which was inappropriate for reasons I have already discussed.

Unfortunately, the whole project then had to be postponed due to a medical problem before that schedule could be rewritten to take in into account the simplified phased model that was to be used. The project was suspended until after the final set of examinations. When it was resumed, there was no strict deadline date for submission for several months due to the afore-mentioned problems. This invalidated the estimated completion dates, so the project continued with me simply moving through the stages of the chosen methodology as quickly as possible without following a specific schedule.

Chapter 5

Working on large development projects in Java

This chapter details issues relating to building large applications in Java. Throughout their time at university, most of the programming that students do involves writing small, trivial programs that demonstrate their knowledge of a few aspects of programming or a few concepts, such as writing a simple lexical analyser or a program to demonstrate polymorphism. There are of course exceptions, but such assignments are often only marked on the correctness of the code, not the quality and there is rarely any need for the student or another programmer to alter that code in the future.

It is for these reasons that building a large application in Java, with many classes and a need for extensibility has proved to be one of the challenges in this project. This chapter details solutions to this problem that have been used as part of the project. Several issues related to the development of large Java applications have been relevant to this project, as outlined below.

Constants and general settings

The use of certain constants throughout ENCAL v2.04 made the code much more readable and helped compensate for the lack of *enumerations* in Java. For example, the several classes that implemented the `OperationsAndDigits` interface could all use the static constants defined within that interface (interfaces are often used to collect together constants in such a manner). Several of those constants represented the operations that are used by ENCAL's representations: `ADD, SUB, MUL` and `DIV`. These constants can be used in classes that implement that interface without even referencing the inteface they are from, which makes the code easier to read.

There were many other constants and settings that ENCAL needed to store, so a `EncalSettings` class was created. This class contains a set of static constants, many of which are used by several other

classes. An example of the use of such constants is the colours of various parts of the GUI; the GUI itself is split over several classes, yet the same colours are used in several parts. To use the same shade of, say, pink 'manually' in each class would lead to *data redundancy* and possibly an inconsistent colour scheme. It would then be difficult to change the colour of all GUI components that used that shade of pink. Therefore, several colours have been made static constants in the `EncalSettings` class. Because of their static nature (*class variables* rather than *instance variables*), these constants can easily be referred to from any other class e.g. `setColor( EncalSettings.PANEL_COLOUR_1)`.

The settings class was also used to store variables that many classes needed access to, such as the current screen dimensions (many screen components are created in proportion to the screen size). In that particular case, there is a private static field to store the screen `Dimension` and static public selector and mutator methods. The screen size is then set from the main Encal class as soon as the program begins executing.

There are several other non-constant settings that are stored in the settings class, many of which are related to the options that the user can select from the ENCAL options screen. To make these settings easier to work with, there are a small set of constants to represent the discrete values accepted by each setting mutator and returned by each setting selector. For example, there are both `LEFT_TO_RIGHT_EVALUATION` and `BODMAS_EVALUATION` constants which are the only values accepted by the method that sets the evaluation method and the only values that the method that 'gets' the current evaluation method will return. The constants, selectors and mutators are all static and publicly accessable, whereas the variable that actually stores the current evaluation method is private. Therefore, the evaluation method can be identified or modified *safely* from anywhere within the ENCAL program.

Many consider the use of global variables in a program to be bad practice. However, this is only because they could easily be modified accidentaly from elsewhere in the system; by keeping the actual location of a variable whose contents need to be known globally private, a variable can be used both globally and safely.

The use of packages

As Java programs grow, so does the number of files, as each public class must be defined in its own file. Even if several public classes could reside within one file, it would still be unadvisable, as large files are often very unwieldy and difficult to manage. After the first couple of iterations of this project design methodology, the number of files in one directory was very large. It became difficult locating and updating a particular file. The solution was to divide the set of files into packages.

Packages are Java's way of implementing class libraries. Each package has its own name and directory. A package's position in the directory hierarchy helps determine its name. For example, with ENCAL v2.04, there is a `repModels` directory, which contains all files which relate to the implementation of the internal model of one of ENCAL's three representations and corresponds to the `repModels` package. Within that directory, there is a `calcModel` directory which contains all files in the `repModel.calcModel` package (all files relating to the calculator internal model).

With the introduction of packages, several changes had to be made to the ENCAL files themselves. First of all, within each file, there must be a statement which informs the compiler of which package the file belongs to. Also, a class cannot 'see' classes belonging to another package unless it explicitly *imports* that package.

There are several other benefits to using packages: "Two classes can have the same name as long as they are in different packages" [6] and having explicit import statements at the top of a class can act as a quick reminder of some of a class' dependancies.

The packages used by ENCAL v2.04 are: `gui`, `globals`, `coreModel`, `repModels`, `repModels.calc` `repModels.dataTreeModel` and `repModels.iconicModel`.

Javadoc documentation

Large programs are very difficult to manage and maintain unless each class is properly documented from within the code. With C++ programs, this was done using standard comments. When programming in Java, comments can be added in exactly the same way. However, change the comment delimiters that are used and there are still useful comments embeded in the code, but these special comments can be read using a tool that comes with the Java Software Development Kit called *Javadoc* [4].

Javadoc is a command line program that takes a Java source file as an argument and then creates HTML documentation for that class and all other non-standard classes that it can find references to from within that file. A 'website' is produced that contains information on all those classes and their public fields and methods, which can provide easily accessable guide to the architecture of the program and the interactions between its components. The text between javadoc comment delimiters (`/** and */`) can be intepretted as HTML, so the javadoc comment for a method could include HTML components such as ordered lists, figures or hyperlinks.

Javadoc documentation can be useful to both the programmer as a overview of the system he/she is working on and to anybody who wishes to gain an understanding of the internal workings of the system sometime in the future. The latter could be because the system is to be extended, a very possible scenario for ENCAL after the end of this project, so it was important that I made full use of Javadoc when creating ENCAL v2.04. The 'website' produced by Javadoc that details the classes of v2.04 has been included on the same CD as the program itself and some example pages have been reproduced in appendix D.

As mentioned in Chapter 1, most programmers with any experience of Java will find the documentation that is produced by Javadoc very easy to understand, as the Java Application Programmer's Interface documentation was produced in the same manner.

Chapter 6

The Design and Implementation of the Basic Graphical User Inteface

---

As stated in chapter 4, the first iteration of the chosen 'phased' methodology involved the design of the basic GUI. Because of a lack of experience with Swing building GUIs prior to the start of this project, the design and implementation of this phase were amalgamated into one step.

Before the start of this phase, user interface design was researched using both the official Java website [16], which contains many tutorials and example programs, as well as a popular book on Java [4], which includes a chapter that covers "Graphics and Java2D" and two others on "Graphical User Interface Components". The full documentation of the Java API was also explored [15].

To begin with, the main window had to be a `JFrame` Swing component. Other Swing components cannot be directly placed in a JFrame; they need to be added to its *content pane*, which is a Swing `Container` object. Within the window, there needed to be several distinct areas, each of which are different colours and contained different GUI *widgits*. These areas are implemented using `JPanel` objects, which are generic containers.

The initial design was that there was an `Encal` class that subclassed `JFrame` and the three representation GUIs would be `JPanel`s that were then added to an instance of the `Encal` class. The program would be run using the `main` method of that class, from which the default constructor was called to create an instance of the program and the main `JFrame`.

As each representation GUI would be quite complex, it was decided early on that they should each be a different subclass of `JPanel` to prevent the `Encal` class from becoming too large or over-complex. These representation GUIs were then added to the main `JFrame` by creating an instance of each of them using their default constructors.

The next problem was to decide on a *layout manager* for the main `JFrame`. According to [4], "layout managers are provided to arrange GUI components for presentation purposes. The layout managers

provide basic layout capabilities that are easier to use than determining the exact position and size of every GUI component. A `GridBagLayout` was chosen, as it allowed components of varying sizes to be added to the container it was managing and could be set up to adjust the layout when the container is resized, although it proved to be very complicated to use correctly.

The use of such a layout manager allowed the screen to easily be resized and the layout to change accordingly, but there was a potential problem with this design: if the main window was made smaller by the user, then nodes within the data tree representation could disappear from view. This could have been a souce of confusion for the intended user, so the decision was made, with the consent of the client, to prohibit the user from resizing the main window. Instead, the window would automatically maximize itself when the program started, which meant that the program would still run using a range of screen resolutions (see chapter 2).

The initial design of the calculator representation GUI

The calculator GUI also uses a `GridBagLayout` to position its components, as it too has a complex layout. The box that displays the full infix expression is a `JTextField` component which has had its `setEditable` method set to false so the user cannot enter text into it.

The calculator itself is mounted on an internal JPanel which uses a `BorderLayout` layout manager. This particular manager allows a component to be added in up to five regions: `NORTH, SOUTH, EAST, WEST` and `CENTER`. The four regions around the edge of the container are as small as their contents will allow, while the central region fills all remaining space in the container. The calculator display box was placed in the top region and the central region was taken up by the calculator keypad. The calculator display box was another uneditable `JTextField`; however, this time, the field had to be set to use right text alignment. The calculator keypad (another `JPanel` uses yet another layout manager: `GridLayout`. This "divides the container into a grid so that components can be placed in rows and columns...Every component in a `GridLayout` has the same width and height" [4]. It was therefore perfect for laying out the keys on the keypad. Each individual key was a `JButton`, some of which had their background colour set.

The look inside panel was created using a `JList` which was then controlled by a `JScrollPane`. The latter allows the user to scroll down the list if it gets too long. Beneath that, there is a `JButton` to toggle its visibility.

The initial design of the data tree representation GUI

This `JPanel` subclass uses a `BorderLayout`; its `EAST` region is a side panel containing three `JButton`s and its `CENTER` region is an instance of a `JPanel` subclass onto which nodes should be drawn. This is another example of using user defined subclasses of Swing components to make working with complex GUIs easier.

The initial design of the iconic representation GUI

This part of the GUI uses a `BoxLayout`, which allows "GUI components to be arranged left-to-right or top-to-bottom in a container" [4]. In this case the panel is split vertically into three. The top panel contains a `JLabel` for each book and the shelf as well as another uneditable window in which expression propagation errors can be displayed. The middle panel is comprised of an array of objects of another `JPanel` subclass, each of which represents a bookcase column. Finally, the bottom panel contains the `JButton` that is used to clear the bookcase.

The well-formedness indicator

Because there can be several yellow faces on the ENCAL GUI at any one time, they have their own class, `WellFormednessIcon`, which is a subclass of JLabel and therefore instances are easy to add to any Swing container. The state of the class can be set to well-formed, not-well-formed or empty using public constants defined within the class. Each instance has a `draw` method within which the face is drawn according to the current state. Finally, the diameter of each object is proportional of the total screen width.

The setup screen

The setup screen was modelled as another subclass of JFrame. Its contents include a large *gif* file (the ENCAL v2.04 logo) on a `JLabel`, the setup options and `JButton`s with which the user can continue to the main screen or exit. The size of the frame is set, so the simplest form of layout manager, the `FlowLayout`, could be used.

Configuration options that have more than two mutually exclusive values were modelled using `JRadioButtons` with `ButtonModel`s to make sure that only one button could be selected at once. Preferences with only two possible values are controlled using `JCheckBox`es along with carefully worded label captions.

The settings class

All of ENCAL's settings then needed to be stored in a way that allowed them to be accessed from anywhere in the program. This was achieved by making each setting's private value and public selector and mutator functions *static* and placing them all together in a `Settings` class. This meant that the selector, for example, could be referenced anywhere (that could 'see' the package `Settings` is in) using the method name preceded by the class name, like `EncalSettings.getBodmasOrLeftToRightEvaluation()`; a reference to a particular `EncalSettings` object is not needed. As discussed in chapter 5, each of these settings has its own set of discrete possible values, which are all implemented as public static constants. This somewhat alleviates the problem of not having *enumeration*s as there is in C++. An enumeration is like a single-field class, except the user defines the discrete set of values that the field can take.

As mentioned in chapter 5, the `Settings` class also contains the screen dimensions and other constants used by the program such as the number of bookcase columns and the details of widely used colours.

Event handling

Whenever a user manipulates a Swing component in any way, an *event* is triggered [4]. Each event contains a small amount information about the interaction, such as the source that triggered the event. An event can be captured by attaching one or more event *listener* to a component.

The software produced during the first iteration of the methodology only features one type of event: the `ActionEvent`. In this case, they are triggered when the user clicks on a `JButton`. An listener object that implements the `ActionListener` interface is attached to all `JButton`s that need monitoring. To implement that inteface, the object's `actionPerformed(ActionEvent event)` method must be called.

As part of this phase, the class for the data tree GUI was made to implement the afore-mentioned interface. Within the `actionPerfomed` method was code to display a message whose contents depended on the source of the `ActionEvent` captured.

The main ENCAL GUI class has an associated private class that also implements the `ActionListener` interface and an instance is added to the setup screen's 'Continue' button, the main screen's setup button and both screens' 'Exit' buttons. Within its `actionPerformed` method, if the source was the 'Continue' button, then the state of all the `JRadioButton`s and `JCheckBox`es is captured and saved in the `Settings` class, then the panel that is to be hid (if any), the look inside panel and the full expressions have their respective components' visibility set. If an exit button was pressed, however, then a box appears prompting the user for confirmation. If either the 'Setup' or 'Continue' button was pressed, then the setup screen is hidden and the main screen is visible (or vice versa).

The only other event handling that was implemented at this stage was to make the button beneath the look inside panel work and to make sure that the events could be captured from the calculator keypad by outputting the semantic value of each key to the console when pressed.

The end of the phase

The final module class diagram using simple UML has been included in appendix E. Very little more could have been added to the GUI at this point without knowing how it would interact with the internal workings. For example, there was no way of knowing what code should be executed when a calculator button was pressed and the bookcase GUI could not be designed without a supporting data structure to hold information about its current state. The next phase was therefore to look at what internal core structures would be needed and how these would need interact with the representation GUIs.

Chapter 7

Design of the internal data structures

Every time the user makes a change to the ENCAL system, there is a need to examine and alter information regarding the current state of ENCAL, whether that be a change that only affects one representation or all three. For state information to be stored and retrieved, at least one internal data structure is required. Initially, it was thought that there were three distinct possible solutions, each of which is discussed below.

A single core structure

All information on the current state could have been stored in one single structure, which would have been queried and manipulated directly by methods within the GUI classes. The main advantage of such a solution was that there is no danger of incorrectly duplicating data and creating inconsistencies. However, this approach was deeply flawed.

Firstly, a single data structure would have to be very complex if it were to store all possible expressions that could be formed by any representation, as would any methods that altered or examined it. This was because it would not just have to store information that is common to each representation, but all information that is particular to each representation too.

Storing so much data was not really necessary either; each representation only needs to know about the current well formed expression or that there isn't one, as only well-formed expressions are propagated throughout ENCAL.

Another disadvantage was that a complex single data structure would make the system very hard to extend in the future, especially if the extension involved drastically altering or even replacing a particular representation. The latter is quite possible, as Dr. Harrop and Prof. Green have both expressed a dislike of the current way the user interacts with the data tree and have several designs for possible replacements (for example, [9]).

Finally, deciding which states of the internal structure were legal in a particular representation would have been extremely difficult.

A data structure for each representation

If there had been a data structure behind each representation, then only the data that was useful to each representation would have been stored. The structures themselves would therefore have been much simpler, so it would have been easier to alter a structure at a later date and decide on the well-formedness, but the methods for mapping between structures would have had to be quite complex. They would have resulted in a strong functional dependancy between the data structures, so altering or replacing a structure so that it could still interact properly with the system would have been difficult, as it would always need to know how the other representations are implemented.

A core structure and a structure behind each representation

What was needed was a level of abstraction; an interface through which allowed the representations to communicate and share data, but did not require them to manipulate each other directly.

My solution is comprised of two parts. The first is a core structure that can only store well-formed expressions, as that is all that the representations should trade in. The second part is that there must be a data structure behind each representation that can store information about the representation in *any* state, whether it is equivalent to a well-formed expression or not.

When in use, the sequence of events is as follows: a representation GUI is changed by the user. If that change alters that representations semantics, then the model behind that representation is changed. This causes the representation model to update the common interface: the core. The core then signals to the other representation models that they all need to update themselves to remain equivalent. Finally, each representation updates its GUI.

When a change is made to a representation by the user that leaves it in a not-wellf-formed state, then the core is set to a particular state that indicates it is empty. The other representations then simply update themselves then their GUIs as before.

This design encourages the re-use of the v2.04 software as there is only functional depenancy between each representation and the core, aided by the simple design of the core, as data structures that can store simple arithmetic expressions are well known and understood. A representation could hopefully be replaced by another without too much knowledge of any other classes than the core, the settings class and the GUI classes.

The interaction of the architectural components

The actual way in which the components described in the previous section interact was almost as important as the data structures themselves. The MVC pattern was used as a basis for the architecture

and way in which the components communicate, as suggested in Dr. Dimitrova's project proposal and briefly discussed in chapter 1.

In ENCAL v2.04, the Model View Controller pattern is actually used in several parts of the architecture. The representation data structures could be seen as being both views and controllers of the core, yet each representation structure could itself be said to be a model and its GUI be both a view and controller of that model. Such a combination of a view and a controller is often called a *delegate* [5], especially when referring to Swing components.

In Java, there are already facilities which are part of the Application Programmer's Interface that can help implement the MVC pattern in a program. The model should be an `Observable` object, which is "the *subject* in the Observer design pattern". Each delegate then needs to implement the `Observer` interface. Each observer must be first be registered with an observable object if it is to be updated when the model changes. Within the observable object, there must be an explicit call to `notifyObservers()` when the delegates are to be informed of a change. This could all be coded manually without the use of interfaces but this would result in excessive functional dependancy or *tight coupling*.

The reason that the Observer pattern was not used in v2.04 is that *all* observers are notified of a change and updated. If, for example, the bookcase updated the core, there would be no need for the bookcase itself to be updated by the core along with the other representations. It could even detract from the usability of the program, as the bookcase would be reconstructed to match the core and so books would be placed on shelves in the default order, which may not have been how they were arranged previously by the user, thus causing confusion.

Instead, a decision was made to design and code the interactions between the representations using interfaced developed specially for this project. First, each representation was made to implement a `EncalRepresentation` interface which contains the following methods: `updateCoreFromRep()`, `updateRepFromCore()` and `setRepGUI( EncalRepGUI repGUI )`. Each representation GUI panel then was made to implement a `EncalRepGUI` interface, which contains a single method, `updateGUI()`.

The sequence of events with this interaction model is as follows: first, each GUI must be registered with its respective model and each representation model must be registered with the core. Then, if the GUI of a representation is altered by the user, the underlying model is changed directly. As a result, the model's `upateCoreFromRep` method is called, which changes the core and passes the core a reference to itself. The core then compares this reference to an internal list of registered representations. It then calls the `updateRepFromCore` method for all representation models other than the one that changed it. These representation models then each call the `updateGUI()` method on their registered GUI. This all results in a communcation system where there is very little coupling between the representations.

Chapter 8

Design of the core data structure

As discussed in the chapter 7, the core data structure plays a key role in ensuring the extensibility of ENCAL v2.04 by only being capable of storing well-formed expressions. Before a particular data structure could be chosen or designed, there was a need to clarify exactly what had to be stored.

A well-formed expression is comprised of two types of entities: operators and numbers. With ENCAL, each operator has exactly two operands, as no unary operators such as unary minus or square rooting are used. Each operand can itself be an operator in another operand-operator-operand triple or it can be a number. The four operators used by ENCAL are add, subtract, multiply and divide. Because these operators are referred to so frequently, they are implemented using constants which are defined in an interface, as described in chapter 5. The range of numbers that can appear within well-formed expressions is limited to just positive integers by what values can be input using the representations.

The type of data structure

One of the simplest way to store such such expressions is in infix form using an ordered list of generic expression tokens, each of which is be either an operator or number. The problem with this approach is that extracting structural information from an infix expression can be very difficult and an evaluation method such as left-to-right or BODMAS is needed to resolve any ambiguities in the expression. For example, an automaton needs to know which operator in $2 + 3 \times 4$ to evaluate first, as it is ambiguous.

An alternative to using a linear structure would be to use something that is defined recursively, such as a binary tree [23]. The previous description of well-formed expressions is highly suggestive of such a structure. The main advantage of such an approach is that the structure of the expression is simply the same as that of the unambiguous tree in which it is stored. All internal nodes are operators and all leaf nodes are numbers (see fig 8.1).

Once the decision had been made to use a binary tree for the core data structure, there then arose the

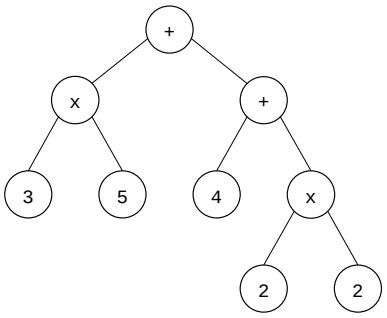

Figure 8.1: The expression $(3 \times 5) + (4 + (2 \times 2))$ represented using a tree

Node data	+	×	3	5	+	4	×	2	2
Node height	3	2	1	1	2	1	1	0	0

Figure 8.2: A tree structure using a only the node height and an ordered list

question of how to implement it in Java.

Implementing a binary tree

Prof. Green suggested that one possible way to represent a binary tree in a program is to store, for each node, its height in the tree and its associated data, then store all of these pairs in an ordered list. The tree of fig 8.1 can then be constucted using such a list (see fig 8.2) because the vertical position of each node is its height and the distance from the left-hand-side of the diagram is directly proportional to the node's index in the ordered list.

The second way is to use the linear expression, as discussed before, as this contains all structural information needed to form a tree, so long as the method of evaluation (left-to-right or BODMAS) is known.

The third way to represent a tree is to use what are referred to in [13] as adjacency matrices. An adjacency matrix for fig 8.1 is shown in fig 8.3.

Each node has a matrix entry $u_i v_i$, with a value of 1 or 2 corresponding to an arc (or directed edge) from node i to node j, while a value of 0 means that there is no arc from i to j. Both 1's and 2's are needed because the former indicates a left-child relationship and the latter indicates a right-child; ENCAL needs to be able to distinguish between them. Note the use of distinct 'from' and 'to' axes, which enable this data structure to support directed graphs, of which binary trees are a subset. It must be possible to uniquely identify all nodes in order that they can be referenced in the matrix, so the nodes in fig 8.3 have been labelled alphabetically in a preorder fashion (preorder traversal involves recursively visiting a node before its left, then right subtrees). The data associated with the tree could be stored in a lookup table based on node labels, or a matrix of node objects could be used instead of the aforementioned matrix of either 0s or 1s.

A similar data structure to adjacency matrices, also discussed in [13] is the adjacency list. Each

		To								
		A	B	C	D	E	F	G	H	I
From	A	0	1	0	0	2	0	0	0	0
	B	0	0	1	2	0	0	0	0	0
	C	0	0	0	0	0	0	0	0	0
	D	0	0	0	0	0	0	0	0	0
	E	0	0	0	0	0	1	2	0	0
	F	0	0	0	0	0	0	0	0	0
	G	0	0	0	0	0	0	0	1	2
	H	0	0	0	0	0	0	0	0	0
	I	0	0	0	0	0	0	0	0	0

Figure 8.3: A tree structure represented by an adjacency matrix

Node label	Left child	Right child
A	B	E
B	C	D
C		
D		
E	G	F
F		
G	H	I
H		
I		

Figure 8.4: A tree structure represented by an adjacency list

node has its own list of adjacent nodes; in this case, these lists will either be of length zero or two, as each node will have either zero or two children. Fig 8.1 is expressed in terms of an adjacency matrix in fig 8.4. Again, a labelling system is required. The data is stored in a lookup table.

Finally, a tree can be represented using a recursive structure, as discussed in [1], [22] and [23]. This approach lends itself well to object orientated programming, as each node is treated as an object. This is often implemented by having a node class which has private fields for the data that needs to be stored at each node and fields for references to its two children (which are of the same class). If a particular node is an operator, then both fields will reference other node objects, otherwise it must be a numerical node and a leaf node; both fields are then null references. The tree is then usually accessed from outside the structure via a reference to its root.

The chosen implementation

The recursive structure was chosen over the other possible implementations because that particular design is so well documented. It appears in almost every book on Java and data structures. The main disadvantage of using a recursive structures is that they require recursive methods to traverse and alter them and the labelling and accessing of a particular node within the tree can be difficult. The latter, however, was not a problem, as with this design, only very simple recursive methods were needed for recursive building of the tree and for recursive traversal. The tree is never modified once built, as it is generated every time a representation model is changed.

At this point in the project, confusion had arisen as to how to use recursive methods and structures, so Kenneth Tait was asked for advice. We met, discussed his work on v2.03 and he demonstrated a small program that illustrates how to use recursion to build and manage expression trees [17]. This helped immensely and progress could again be made with the design of the core of v2.04.

The core of ENCAL v2.04 is comprised of three classes and one interface and is outlined using simple UML in appendix E. A single node class would not have been sufficient as the data field of each node cannot store either a number or operator as operators cannot be of the same basic type as numbers. For example, a constant `ADD` could not be defined to be of type `integer` with value `1` and all numbers also be stored as integers because then the value `1` would be ambiguous. Instead, a `CoreNode` interface was used which defines the operations that all nodes must implement. There is also a `CoreOperatorNode` class which has a `char` field to store an operator and fields for both the left and right child nodes. Each leaf node is of the class `CoreNumberNode`, which has a `float` field to store the node's value, but it does not have any references to children, as a number node should not have any children. The `float` type was used so that the system could be extended to use real numbers, if so desired. Both of these classes implement the afore-mentioned interface. The tree is accessed through the `CoreRoot` class, in which all fields and methods are static, so it can be used to store a reference to the root of the core tree which can be accessed from outside the core without needing a reference to an instance of the `CoreRoot` class.

The methods that are included in the `CoreNode` interface include a method to evaluate a node and (if it is an operator node), its subtrees. The divide by zero anomaly results in an exception being thrown, but it is not caught within the core; instead it is allowed to propagate back to the user interface so that an error message box can be shown. Another `CoreNode` method returns the maximum *height* of the node, which is the maximum distance to any of its descendants. There are also methods to output the node's value as a `String` and to output the node's subtree as a `String` (using an infix traversal), which was useful for testing the core module.

The `CoreRoot` class contains a private static reference to a `CoreNode` and an associated selector and mutator through which the rest of ENCAL can access the core. It also has a list of registered representations (see chapter 7). The root reference mutator requires two parameters: the replacement `CoreNode` reference and a reference to the representation that is requesting the change of root. The mutator changes the root, then iterates through its list of representations, updating all apart from the one

that made the request.

Also included in the class is a method to clear the core, which simply calls the afore-mentioned mutator, supplying it with a null reference rather than a `CoreNode` object. It is when clearing the core that the importance of Java's garbage collection comes into play. When the core root reference is made null, then there are no references to its children from within the rest of the program, so they are marked for garbage collection, then *their* children are marked, etcetera. The Java Virtual Machine provides no guarantee that garbage collection will be performed, but ENCAL does not require constant data processing, so there is plenty of time inbetween user interactions in which the garbage collector can run. As a result, many of the memory-related problems with v2.03 are avoided.

Core module testing

Once implemented, each method of the core module was tested to an extent, but thorough testing was made difficult by the recursive nature of the test module. More extensive testing was performed when the core was attached to the implementation of the first representation to be designed: the calculator representation.

Chapter 9

The calculator model

The third iteration of the phased methodology involved the design of the design of the calculator representation data structure. This needed to be able to store any expression that can be formed using the controls on the GUI or that it can be passed from the core, from something as bizarre as $3\times+-\div4+((+3$ to something straight forward and well-formed, such as $3+(2\times4)$.

The functional requirements of the representation can be summarised as following:

- The user must be able to add an operator, digit or parenthesis to the current expression in infix form.
- The user must be able to undo the last addition made to the current expression in infix form.
- It must be possible to clear the expression completely.
- It must be possible to evaluate the core.
- It must be possible to update the representation from the core.
- It should be possible to display each operator-operand-operator-result quadruple, although this is not essential.

As several of these requirements refer to the current expression in infix form, it was decided that the calculator model be a linear structure based around this form.

The most basic units used by this structure are *expression tokens*, each of which can be either an integer, an operator or a left or right parenthesis. This design was inspired by Kenneth Tait's demonstration program, in which tokens are used and modelled in a similar way [17]. Tokens are implemented using an abstract class, `Token` and three subclasses, `NumberToken`, `OperatorToken` and `ParenToken`. The only method that is inherited from the abstract class is a method to return the precedence of the token. This is discussed in more detail later on.

The whole calculator model is centered around the `CalcModel` class, which implements the `EncalRepresentation` interface. Because there should only ever be one instance of the model in existance at any one time, the model was made to implement the Singleton pattern [6]. This pattern specifies that the class must contain a field which is a private reference to an instance of itself and a public static access method which either returns a reference to the private field if the field reference is not null or creates an instance of the class to store in that field and returns a reference to that. Other static methods and fields are then not necessary to ensure that only one instance can exist at once and all of the class' methods are called on the reference that is returned by the afore-mentioned static `getInstance()` method.

Updating the core from the calculator model

The model is focused around a list of `Token`s which represents the current expression in infix form. The first step in propagating from this list to the core is to decide whether or not the list represents a well formed expression. A method was devised to check this, which uses the following rules:

- The number of opening and closing parentheses must be equal
- There must be no adjacent operators
- An opening parenthesis cannot be preceded by anything other than an operator or another opening parenthesis
- Adjacent numbers are not allowed
- A number is not allowed to follow a closing parenthesis
- A closing parenthesis can only follow a number or another closing parenthesis
- The last token must be either a number or a closing parenthesis (if the infix expression is not 'empty')

If the current infix expression does not pass the well-formedness check, then the core is cleared. If it does pass, then it must be used to update the core.

Building the core directly from an infix expression is difficult because of the ambiguous nature of infix, the use of parentheses and the difficulty in extracting the structural information needed to form a tree. To get around this problem, when updating the core from an infix expression, an intermediate form of the expression is used. This form is called *postfix* or *Reverse Polish Notation*.

Postfix differs from infix in that each operator is preceded by its two operands rather than being inbetween them. This small change in notation means that parentheses are not needed and expressions are no longer ambiguous. For example, $2 + 3 \times 4$ is ambiguous, but with a postfix expression such as $34 \times 2+$, there is no question as to which operands are associated with which operator and which part of the expression should be evaluated first.

The process to convert from infix notation to postfix is a well documented one. It involves the use of what is often referred to Dijkstra's Shunting Algorithm [19]. A version of the algorithm has been included in appendix C. The algorthm uses two stacks: one as temporary storage for operators and parentheses and the other with which the postfix expression is built. The algorithm requires that a precedence be associated with each `Token`; this is where the afore-mentioned abstract superclass method comes in. The usual shunting algorithm rules are that opening parentheses have a lower precedence than addition/subtraction and which in turn, have a lower precedence than multiplication/division. The algorithm usually only converts to postfix using the BODMAS rules, but it was modified for this project so that left-to-right evaluation is also catered for; within the algorithm, the `EncalSettings` class is checked to see what the current evaluation method is. To convert to postfix using left-to-right evaluation, the rules are changed so that multiplication, addition, subtraction and division all have the same precedence.

Once in postfix form, the core can be updated using a recursive method that returns a `CoreNode` object. Within the method, if the last token is a number token, then create and return a core node using the token's value. Otherwise, the last node must be an operator token, so a core operator node is created using the token's value. Its right child is then set to the core node returned by calling the recursive method on the current token list minus the last element. The same is then done for the left child.

Finally, `updateGUI()` method of the `EncalRepGUI` that is registered with the calculator model is updated.

Interaction with the GUI

The two ordered lists of `Token`s used by the calculator model (infix and postfix token lists) are implemented using a `TokenList` class, which contains a vector for containing the tokens in order and methods to access and manipulate that vector. As well as having a method to append a token, there is also an 'append and concatenate' method, which is called whenever the user presses a calculator keypad key. This is needed because of the need to concatenate adjacent number tokens in the infix expression, as discussed in chapter 2.

If the 'undo' button is pressed and the infix expression is non-empty, then the last infix token is removed and the core and GUI are updated.

Methods were added so that the model could evaluate the core root (and therefore the whole core tree) and the result could be displayed in the calculator display whenever the '=' button is pressed. It is at this point that 'divide by zero' exceptions are caught; a message box appears on screen and tries to explain the divide by zero anomaly using an example that the inteded users will hopefully understand.

Another part of the necessary interaction between the GUI and model was that the calculator display should always show the value of the last number token in the infix expression until the equals button is pressed. A method was written to return the numerical value of the last number token in the infix token list.

Within the `updateGUI` method of the `CalcPanel` class, the infix expression text field is updated,

as is the yellow face that indicates well-formedness, through a static access method of the calculator model class.

The only part of the GUI/model interaction not to be completed sucessfully was the implementation of the 'look inside' panel. The design was attempted, but a particular bug could not be solved. Rather than copious more time on an unessential feature, the project was advanced to the next phase. Unfortunately there was never the time to return and fully implement this feature.

Updating the calculator model from the core

When the core calls the `updateRepFromCore()` method of the model, the infix expression is immediately cleared. Then, if the core root reference is not null, a recursive method that takes a `CoreNode` as a parameter is used to perform an infix traversal on the tree and build the infix token list. The recursive method is first passed a reference to the root. If it is a number node, then it is appended to the list. Otherwise it must be an operator, so an opening parenthesis is appended, then the method is called again on the node's left child, then an equivalent operator token is appended, then the method is called on the right child before finally appending a closing parenthesis. After the method has terminated, the registered GUI is updated.

Testing the calculator module

The testing of the module was perfomed with the help of a `CalcModelTest` test driver, which allowed the calculator GUI, the calculator model and the core to all be tested as a whole. The core could at last be tested thoroughly, as there was an interface through which data could quickly be fed into the system. After a considerable time was spent ironing out minor bugs, the module passed all tests apart from those relating to the look-inside panel.

Chapter 10

The data tree model

The design of the data structure behind the data tree model was the fourth iteration of the chosen design methodology.

The data tree internal structure

The data tree representation, when in a well-formed state, closely mirrors the core representation in that they are both tree structures. However, the data tree model could not have used the same data structure as the core, as it must store much more than just well-formed expressions. It must be able to cope with storing binary forests, as more than one disconnected node can exist at once and it also must be able to store operator nodes with only one child. It must also be possible to store 'blank' nodes. A different way of storing a directed graph was needed.

The alternative ways of storing binary trees from chapter 8 were considered for the data tree model. The most appropriate of these was the adjacency matrix, as it can allow multiple roots. Such a matrix could be built using a simple traversal of the core.

An adjacency matrix could be checked for well-formedness by an examination of each row (if the parent nodes are listed down the vertical axis of the matrix). If the row corresponds to a operator node, then it must contain exactly two entries which refer to children. However, if it corresponds to an integer node, then it must have no references that refer to chilren. Cyclic graphs can be detected, which are not easily to find with recursive tree structures, by counting the total number of references to children. This should be one less than the total number of nodes if the graph is non-cyclic and connected.

An alternative design could involve using a similar structure to the core, but changing the class that is equivalent to the `CoreRoot` class so that it can store a list of roots. The operator node class would be altered so that zero to two child references could be stored. The problem with this design is that it is difficult, as mentioned before, to locate a particular node in a recursive structure. This locating would

be necessary if two trees were joined by the user with an arc.

Change of plan: the simplification of the data tree representation

At almost every meeting throughout the course of the project, Dr. Harrop and Prof. Green would present new versions of the data tree that they had devised. They had both said on several occasions that one of the main reasons for wanting an extendable version of ENCAL is that they wish to replace the existing data tree representation with one of their new designs.

Shortly after begining designing a replica of the data tree representation of ENCAL v2.03, it was realised that was time was running out. This could have been due to the lack of a strict schedule at this point. It was decided that the best thing to do given the circumstances was to not implement all the functionality of the data tree representation and concentrate all efforts on the representation that would not be getting upgraded so soon i.e. the iconic representation. The decision was made to implement the methods that update the representation and draw the data tree on screen, but not allow the user to interact in any way with the data tree. This saved lots of time ime as both myself and my supervisor considered the user interactions with the data tree to be potentially quite difficult to design and code.

The design of an uneditable data tree representation

As the data tree was to be uneditable, there was no need to store anything in the data tree model other than a reference to the core root. The `DataTreeModel` class implements the necessary `EncalRepresentation` interface and uses the Singleton pattern like the calculator representation, but includes very few other methods.

The important part of the design is the `DataTreeInnerPanel` class, which is updated when the main data tree panel is updated. It overrides the standard Swing component `paint` method to draw the tree using Java2D features. Within that method, there is a call to a recursive method that recursively draws the core's nodes. This inner method takes six parameters: the `Graphics` object onto which to draw, the `CoreNode` to be drawn, the x coordinate of the node, the horizontal offset of the node's children from the node, the vertical position of the node and the vertical spacing between levels in the tree. During each iteration of the method, a node is drawn (node rectangle, outline and value), then, if it is an operator node, the arcs are drawn to where its chilren should be, then the children are drawn. The method is initially passed the root.

Unfortunately, due to a bug somewhere, the arcs are all slightly offset nodes from the nodes. A solution was not found to this problem.

Finally, because the user is prohibited from interacting with the data tree, the buttons on the data tree side panel were set up so that when one is clicked it brings up a message box explaining why that particular feature is not necessary.

The testing of the data tree module

Once the data tree module had been implemented, it was worked into the software that was produced for the previous phase. It works well with the core and calculator representation. The only tests that it did not pass involved the problem with the arcs and nodes not connecting and also that numbers are displayed from just to the left of a number node's centre out to the right. This meant that a large number are displayed with only the left-most few digits inside the respective node.

Chapter 11

The iconic model

The design of the iconic representation was the last phase of the chosen design methodology. As the data tree model was made relatively simple due to limited time, the iconic model is the most complicated of v2.04's three representations.

A comparison of two possible designs

Within the iconic representation, there are four columns, each of which can contain up to six shelves. Each shelf can hold up to nine books, each of which must be either blue or green. An object-orientated approach to modelling such a system often focuses on the adjectives and nouns used to describe it. An examination of the description above suggests the classes, relationships and multiplicities shown in fig 11.1.

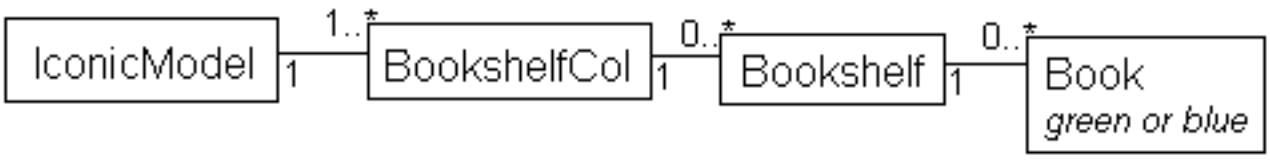

Figure 11.1: Possible iconic model class outline

Another possible design was also considered: the bookcase could have been modelled as a 3D matrix of entries which could each correspond to either a blue, green or empty book slot. This design was quickly discarded, as it could not have distinguished between an empty shelf slot and an empty shelf. A second matrix would have been needed to store the positions of all empty shelves, but this

design could have resulted in discrepencies such as a book being found in an empty shelf slot. The previous object-orientated approach is much more suited to programming in Java and allows tighter retrictions to be placed on the way it stores its data.

The chosen iconic model design

The object-orientated design can be seen as a hierarchy of iconic components. The ordering of an iconic component's sub-components is important, so each class of iconic component apart from the Book class has an array of sub-components; null references within such arrays are used to indicate empty book or shelf slots.

At each level of the hierarchy (apart from the Book level), methods are required to select individual sub-components so that functions such as the recursive drawing of the model can be perfomed. With Bookshelf and BookshelfCol objects, methods are needed to add and remove subcomponents and so both types of objects need to know when they are either 'empty' or 'full'. The proportions of the bookcase in terms of its capacity and its sub-components capacity are stored within the `EncalSettings` class.

Once the model had been designed, there was then the task of integrating it into the system. The first step was to ensure that only one iconic model can exist at any one time by applying the Singleton pattern to the `IconicModel` class. Next, that class was made to implement the `EncalRepresentation` interface and its associate gui, the `IconicPanel` class was made to implement `EncalRepGUI`.

The updating of the core from the representation was made possible by writing a method for each class of iconic component which returned a reference to a tree of `CoreNodes`. Each of these trees are semantically equivalent to the component that built them; a traversal of the iconic hierarchical structure is used to amalgamate all these trees to form a new core.

Updating the representation from the core proved much more difficult. A recursive approach was investigated, but it would not work with the non-recursive iconic structure. The problem is that the height of each node cannot be used reliably to work out which subtrees refer to which iconic column because of the way that the height of a shelf subtree and a column subtree can vary. The full functionality of this method was not achieved; instead, only expressions of the form x, $x+y$ or $x \times y$ can be propagated to the representation. All other expressions result in the core being cleared.

The `updateGUI` method of the iconic panel requests a repaint of each of its `BookshelfColPanel`s. The iconic structure is then iterated through so that each shelf and then its books are drawn. The visual positioning of each component is achieved via methods that convert between model coordinates (book number, shelf number, column number) and standard screen coordinates.

The updating of the model by the GUI was another area where full functionality was not implemented. Mouse event listeners, similar to the action event listeners used throughout the GUI, were added to the icons which are to be used to add books/shelves. Shelves can be added by releasing the mouse button over a column when in possession of a new shelf; this was implemented by seeing if the coordinates at which the button was released are within the bounds of each screen column. This

is possible because each mouse event captures the coordinates at which the event occurs. If a un-full column is found that contains those coordinates, then a shelf is added to the equivalent column in the iconic model and the GUI and core are updated.

The intention was to add books using a similar method. However, there does not seem to be a simple way of finding a bounding rectangle for each shelf, as the mouse event is detected from within the iconic panel class not the `BookshelfColPanel` class. The size and position of each shelf's bounds could have been computed using the proportions of the iconic structure and the bound of each column's panel. However, this could not be achieved because the coordinates of the mouse events are captured are relative to the iconic panel and the offset between the iconic panel and each column's panel could not be found. Due to both these problems and the approaching project deadline, books cannot be added to the iconic representation using the iconic representation GUI, even though the mechanism to add books to the iconic data structure is functional. In v2.04, books cannot be moved or deleted and shelves cannot be deleted, for the same reasons.

Testing the iconic module

Before the iconic structure was attached to the core and its GUI, it was tested by itself and passed all tests.

Its interaction with the whole of ENCAL were then tested; the extent of the testing was limited by the simplistic method with which the representation is updated from the core and by the lack of a way to add books using the representation GUI. The only two ways in which the propagation of data from the iconic representation to the core could be tested was through adding shelves and by clearing the representation. Features such as selecting whether to evaluate books or shelves first worked as desired for the expressions that could be propagated to the representation, but no guarantees could be made that they would function correctly if a more complex expression were to be used.

Chapter 12

Project evaluation

Once the final iteration of the phased methodology used in this project had terminated, the next task was to evaluate the two parts of the project: the product and the process. The product evaluated, v2.04, has been included on a CD with this project, along with the Javadoc documentation files that were produced

The evaluation of the product

The product was mostly evaluated by Dr. Harrop and Prof. Green. They were provided with a list of criteria with which to evaluate v2.04 and they were also asked to comment on their satisfaction with the project in general. Their comments on the individual criteria are included in appendix G; their general opinion is provided below.

> The project undertaken by Mr Furnass was to re-implement a CAI system called ENCAL in Java, from its existing version in Toolbook, with the aims of (1) making it feasible for other people to extend the system by providing clean, well-structured, well-documented code, and (2) making it faster and more reliable. We suggested that he should use an architecture in which a core system represented the state of the world, interacting with any desired number of particular forms of on-screen representation and manipulation.
>
> We address (1) first.
>
> We were immediately impressed when we saw the new version, and we have found no reason to change our extremely favourable view. The new implementation looks very like the old. It handles in the same way, the controls and settings have been faithfully recreated, and the result is much faster. It seems to be completely reliable, whereas the original Toolbook version always crashes after a while, presumably because of memory leaks.

The new implementation uses exactly the architecture we hoped for. That has been a success: almost all the features of the original version have been recreated, although there are some limitations.

We are not in a good position to evaluate the degree to which the code can be taken over by somebody else, but we've done what we can. We've looked at some of the code files and they seem to be clean, well-organised, and well commented, which makes a good start. The JavaDoc documentation is comprehensive, as far as we can tell. The architecture specifies what a new interface (e.g. a new tree editor) must do in order to communicate with the core. This looked entirely comprehensible and quite encouraging. There's also a bit of info about the underlying core model but not a lot, presumably there'll be more in the write-up.

Our estimate is that extending the new implementation would be very much easier than extending the previous Toolbook version. Clean interfaces are provided so that a new representation, such as a new tree renderer and editor, could easily replace the one provided by Mr Furnass.

Overall, although the re-implementation does not fully recreate the original, it has been remarkably successful. The omissions are due to lack of time rather than design failures, as far as we can tell.

The checklist provided by Mr Furnass was very helpful to us in evaluating the degree to which the new implementation met the specification.

The underlying question remains whether it will be possible for another programmer to take the work forward. That depends on the documentation, which we cannot completely evaluate, although what we see looks very good.

Evaluation of the extendability of 2.04

ENCAL v2.04 has a core which is based around a binary expression tree, a common and well-understood data structure. This structure is not dependant on any representation, so a representation could be replaced with another without too much difficulty. As all classes are documented using Javadoc and an outline of the program's workings is included with this report, it will hopefully not take a programmer long to understand the design of the core and the way in which the representation models and their GUI's are linked to it.

V2.04 is less extendable when it comes to altering the code for an individual representation. This is particularily true of the iconic representation, as it uses an unconventional data structure and several of its methods are incomplete, such as the method to update the structure from the core. However, the data structure used is not particularily complex and all methods whether fully functional or not are documented using Javadoc. Therefore, it may not take a proficient programmer very long to complete the functionality of the iconic representation.

Evaluation of the process

The quality and quantity of the background reading Computer Based Learning was not researched as part of this project as the task in hand was concerned solely with software development. Several meetings were scheduled with the clients so design issues could be resolved by the clients. The research into Java and Swing was important for the progress of the project, but several problems with the GUI of 2.04 may have been resolved had more time been spent looking at Swing. Background reading on data structures was quite addequate, but recursion should have been investigated more closely, as too much time was then spent researching this topic when in the middle of designing and programming the system.

Researching methodologies and the selection of a particular methodology Too much time was spent researching specific methodologies that were too complex and 'heavy' for this project; much of the research into such methodologies was omitted from the report because it was so irrelevant. The original choice of the waterfall model was slightly naive, as the model does not allow unexperienced Java programmers to begin small and incrementally build software in order for them to discover the possibilities of programming in Java. The change to the phased model was a wise one, as it did allow incremental building and could be modified if a need for more phases was discovered. The latter proved important when a design for the internal workings was settled upon, as it effectively meant that what had previously been one phase had to be split into four to allow the core and each representation to be designed and built incrementally.

The understanding of v2.03: which features to retain and which to change The older version of ENCAL is understood to quite a detailed level, as can be seen from the description of the system in chapter 2, but the actual capture of requirements was not really adequate. This was because the desirability of each feature which was to appear in v2.04 was not quantified and recorded, which made the evaluation of whether v2.04 has "minimum functionality" difficult.

The quality of the analysis of the problem The problem was well understood because of the way it was broken up into phases; this made the analysis a fairly logical process. The design followed the same structure.

Which project objectives were met; what project extensions were implemented Nearly all the objectives were met with a few exceptions. Firstly, white box testing was not always possible due to the complexity of the system; to compensate, more extensive black box testing was used than was originally planned. Secondly, UML diagrams were not drawn for every software component and the user's guide (appendix F) only supplies instructions on how to install the software, not how to run it. Both of these exceptions are the result of time restrictions.

Only one project extension was implemented as meeting the minimum requirements turned out to be much more difficult than was originally thought. This was the dynamic linking of the representations, a feature that was taken into consideration from the early stages of designing the project. This meant that it never had to be ‘worked into’ the design at a later point in time, which could have been problematic and lead to ‘buggy’ software.

Bibliography

[1] Duane A. Bailey. *Java Structures: Data Structures in Java for the Principled Programmer*. McGraw-Hill, second edition, 2003.

[2] Kobe Davis. *Requirements Engineering in Agile Approaches*. http://pages.cpsc.ucalgary.ca/ davisk/613/Agile.html Last accessed: 09/10/03.

[3] John Davy and Karim Djemame. *SO11: Introduction to Programming (2)*, 2001. Lecture notes for a University of Leeds computing course.

[4] H.M. Deitel and P.J. Deitel. *Java: how to program*. Prentice Hall, fourth edition, 2002.

[5] H.M. Deitel, P.J. Deitel, and S.E. Santry. *Advanced Java 2 Platform: how to program*. Prentice Hall, first edition, 2001.

[6] Nick Efford. *SO21 aka COMP2650: Object Oriented Programming*, 2001. Lecture notes for a University of Leeds computing course.

[7] Andrew Harrop. *Outline of PhD research*. http://www.cbl.leeds.ac.uk/ andrewh/Mywebsite.htm Last accessed: 29/09/03.

[8] Andrew Harrop. *The Design of a Computer-Based Pedagogy For Teaching Calculator Representations*. PhD thesis, University of Leeds, 2001.

[9] Andrew Harrop. *The Second Level of ENCAL*. Technical report, University of Leeds, 2002.

[10] Sun Microsystems Inc. *Javadoc 1.4.2 Tool*. http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/ Last accessed: 28/09/03.

[11] Phillipe Kruchten. *The Rational Unified Process: an introduction*. Addison-Wesley, 1998.

[12] Timothy C. Lethbridge and Robert Laganiere. *Object-Oriented Software Engineering: Practical software development using UML and Java*. McGraw-Hill Education, 2001.

[13] Kevin McEvoy. *CO12 aka COMP1360: Fundamentals of Computer Science*, 2001. Lecture notes for a University of Leeds computer science course.

[14] Jennifer Preece, Yvonne Rogers, and Helen Sharp. *Interaction Design: Beyond Human-Computer Interaction*. John Wiley & Sons, 2002.

[15] Inc. Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification*. http://java.sun.com/j2se/1.3/docs/api/ Last accessed: 30/09/03.

[16] Inc. Sun Microsystems. *java.sun.com*. http://java.sun.com/ Last accessed: 30/09/03.

[17] Kenneth Tait. A small demonstration program written in java. Personal communication.

[18] Kenneth Tait. *Encal 2.03 - a brief reflection*. Personal communication.

[19] Dr. Joe Traylor. Help page for cs-ers. http://web.bvu.edu/faculty/traylor/help_page_for_CSers.htm Last accessed 11/10/03.

[20] Kathy Walrath and Mary Campione. *The JVC Swing Tutorial*. Addison-Wesley, 1999.

[21] Bruce E. Wampler. *The Essence of Object-Oriented Programming with Java and UML*. Addison-Wesley, 2001.

[22] Mark Allen Weiss. *Data Structures and Algorithms in Java*. Addison-Wesley, 1999.

[23] Niklaus Wirth. *Algorithms and Data Structures*. Prentice-Hall, 1986.

Appendix A: Personal reflection on the project experience

The final year project experience was a challenging one, as it required knowledge of many aspects of computing and computer science but went certain areas in much more depth than in any university computing modules. It provided an interesting opportunity to combine the theoretical and practical, which was not covered in much detail in any particular module.

The project experience also proved to be problematic and rather long, as a medical problem meant that the project was finally completed just over a year after it was begun. The problem, diagnosed by an orthoptist as "symptom-producing heterophoria of the non-specific type", meant that looking at computer screens for even relatively short periods of time caused an acute aching of the eyes, headaches and tiredness. For that reason, the project was stopped in the spring and then resumed after the second set of final year exams. This made the project rather disjointed, as much of the work that was done for the first half of the project had been all but forgotton by the time the project resumed on 16/10/03.

Regarding individual parts of the project, the implementation of the graphical user interface proved to be quite difficult. Many aspects of Swing were found to be unpredictable and due to their nature, problems relating to the GUI were often undetectable until the program was run. In particular, choosing the right layout manager to implement a particular GUI design was quite difficult; the more complex layout managers were often described in terms of their features in such a way that they were chosen over the simpler layout managers. With hindsight, it has been realised that the easiest way to produce a GUI is to use many nested `JPanel`s, each with its own simple layout manager. It is then much easier to adjust the layout of each GUI component. Were this project of this nature ever to be attempted again, then I would spend time getting more aquainted with Swing first, as it is a large and complex system that is difficult to learn how to use whilst building a large piece of software under pressure.

Another challenging aspect of this project was the use of recursive structures and methods to design and maintain v2.04's core data structure. A long time was spent with pen and paper trying to form a design that used recursion; eventually, after some helpful pointers from Kenneth Tait, an understanding was developed of how such a structure could be used as part of a real program. This was interesting

because many of the university modules had just looked at the theoretical aspects of recursion and recursively defined structures while others simply looked at software development; very little evidence had been seen of the combining of both of those aspects of computing. I would encourage anyone who is struggling to implement recursion as part of their program to persevere, as the result is often code that is very concise and, once the concept of recursion has been grasped, very readable. For example, the recursive methods used by the calculator representation to update itself from the core is easy to understand and seems to be bug-free. On the other hand, the equivalent method used by the iconic representation is not recursive and as a result is long and complicated.

The project process was an introduction to working reasonably sucessfully on a large development project. However, several problems were encountered. The first was the lack of a schedule for the second half of the project meant that there was no way of knowing if progress was being made at the right rate. This resulted in the last few weeks of the project being slightly rushed.

Another process-related problem was that the evaluation of the process was made harder because very few of the objectives and minimum requirements defined at the start of the project were quantifiable, so stating for sure whether they had been met was difficult. The correct way to approach the requirements analysis would have been to not just include a list of all the features of v2.03 that v2.04 must possess and what additional functionality is needed, but exactly how desirable and essential each of these features is to the client. This would have made a minimum requirement that stated ”the program must have minimum functionality” more easily quantifiable.

The selection of a design methodology was also problematic, as it was required before the system had been fully broken down into its sub-components. The lesson learned from this was that a methodology must be chosen that can adapt as the design is explored; new phases may be needed and deadlines may change.

Overall, the project was very interesting and although there were many problems and not all the desired functionality was implemented, there were several parts which I consider were well implemented, such as the calculator and the extendable core. This project has hopefully provided a solid foundation for anybody wishing to continue developing ENCAL in Java.

Appendix B: Original Project Schedule

Below is listed the schedule that was included as part of the mid-project report. For the reasons given in chapter 4, it was irrelevant almost as soon the mid-project report was submitted and the internal design of the new version of ENCAL was considered. For the most part, each stage was designed to tackle one particular project objective. The acronym 'ECD' is an abbreviation for *Estimated Completion Date*.

- **Stage one: ECD 14/01/2002**
  This stage will be complete when I have a detailed specification of the current version of ENCAL, complete set of use cases and a list of non-functional requirements. Also, by the end of this stage, I will need to have compiled a list of all the features of ENCAL that are not essential, those that can be altered and I should have decided which of the project enhancements mentioned above I am actually going to implement. I have installed ENCAL and Asymetrix Toolbook II Instructor 5.0 on my own computer, so I will be able to complete this stage at home over the Christmas break.

- **Stage two: ECD 01/02/2002**
  In this section, I will need to decide on which software design methodology I am going to use and then refine the use case model that was formed in stage 1. The resulting model of the system will have no ambiguities or redundancy and should provide an internal view of the system and will be complete when I have a complete model of the system.

- **Stage three: ECD 01/03/2002**
  This stage can be split up into two main tasks, one of which follows the other. The first is to split the model up into manageable, distinct components that can be coded and (at least partially) tested separately. The second is to create an object model of my proposed system. The stage will be complete when I have a complete UML class diagram of my system.

- **Stage four: ECD 14/03/2002**
  For the fourth phase of my project, I will need to implement the components formed during the

previous phase in Java and then devise white box test plans for each of them before making sure that they all pass the tests. This stage will terminate when all components have been implemented.

- **Stage five: ECD 07/04/2002**
  At this stage, I will compile all the individual components of stage four to form a complete application. I will then devise a fairly comprehensive black box test plan and proceed to test my program. This stage will be complete when I have checked how my program reacts under every test case circumstance, I am happy with the number of cases that my program does not comply with and I have changed all the code I need to as a result of my testing.

- **Stage six: ECD 01/05/2003**
  This stage involves the completion of the writing of my report. This can be thought of as being in three parts, the first of which is the evaluation of my project: does it meet the minimum requirements, have all use cases been considered etc. Secondly, I will need to include a section in appendix A in which I will reflect on the project experience as a whole and discuss the lessons learnt. This stage will be complete when the report is completely written.

Appendix C: Algorithms

Dijkstra's shunting algorithm

This is used to convert an expression in infix form to the equivalent postfix. Digits have a precedence of 0, parentheses have a precedence of 1, plus/minus of 2 and multiply/divide of 3 if BODMAS is used, 2 if left to right evaluation is used.

```
while there is more of the list to process
   get the next token
   if the token is a variable
      output token
   if the token is (
      push token on stack
   else if the token is one of *, /, +, -
      while the precedence of the token <= precedence of the top
      item of the stack (if any)
        pop the stack and output the token just popped
      endwhile
      push the new token onto the stack
   else if the token is )
      keep popping tokens off the stack and outputting them, till
      you pop (. Don't output either the ( or  the ).
   endif
endwhile
while there's still stuff in the stack
  pop the top token off the stack and output it
endwhile
```

Appendix D: Javadoc screenshots

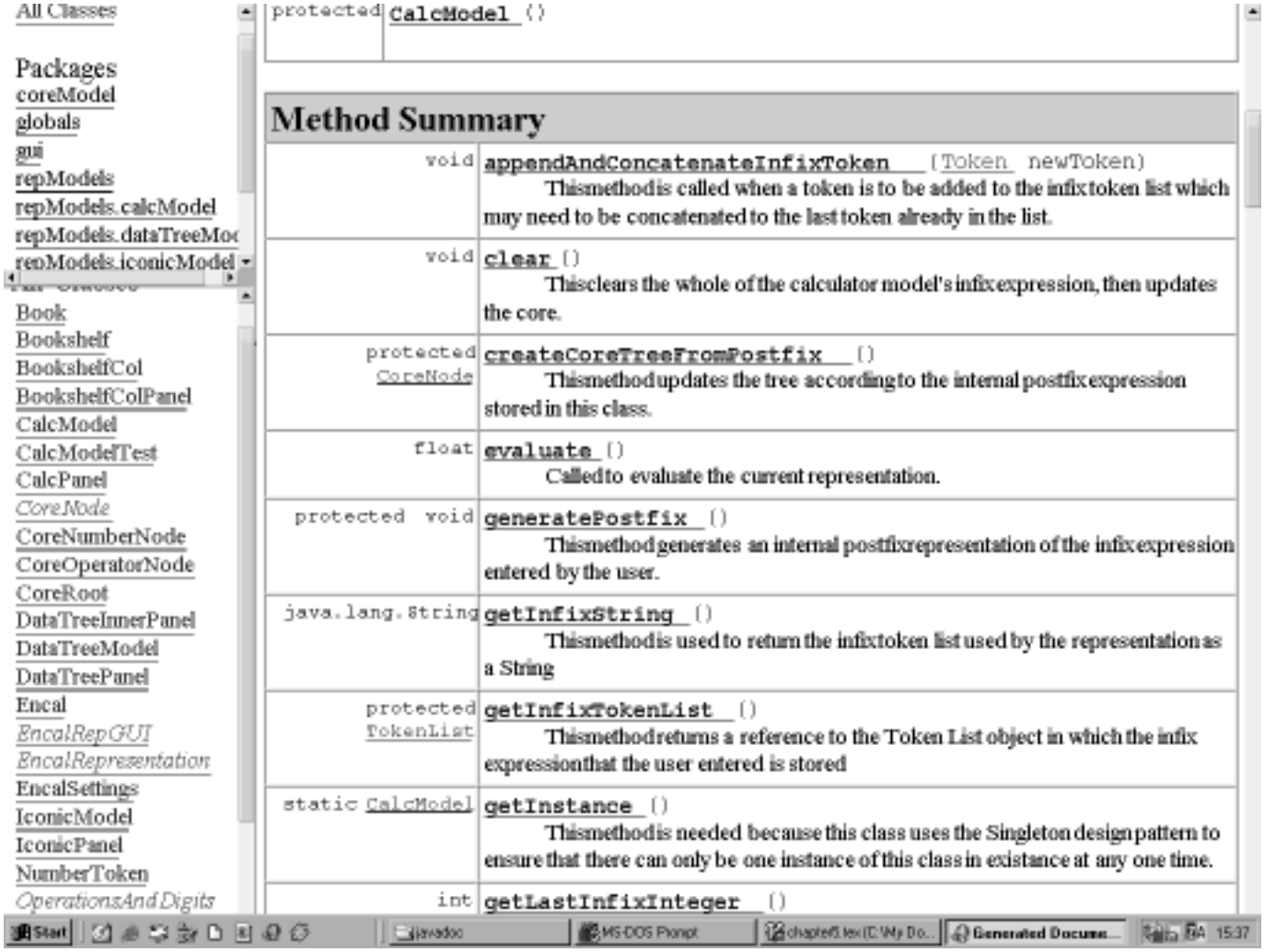

Figure 12.1: Part of the Javadoc documentation for the `CalcModel` class

Figure 12.2: Part of the Javadoc documentation for the `Bookshelf` class

Appendix E: UML of certain modules

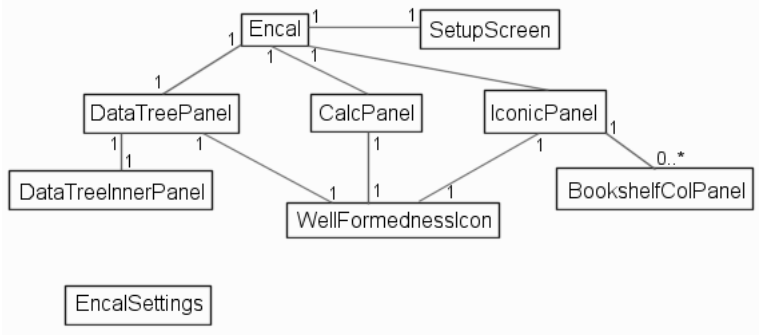

Figure 12.3: The class outline of the basic gui (first iteration of methodology)

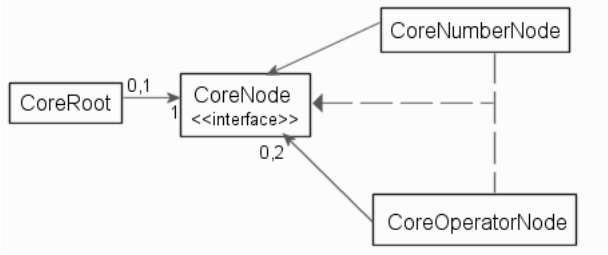

Figure 12.4: The class outline of the core

Appendix F: User manual for ENCAL v2.04

System requirements

ENCAL v2.04 requires Java2 version 1.3.1 or later to be installed on the machine that it is to be executed. Version 1.3.1 is included on the CD provided with this project.

Instructions for running ENCAL

The `PATH` variable that is used by the Java Virtual Machine to locate the Application Programmer's Interface must be set correctly before ENCAL can by run. Please refer to the Java website [**?**] for information on how to do this on your particular computer.

You must now find the ENCAL directory on the CD. If you are running the system on Windows, then you can type the following from the command prompt ('D' should be replaced if necessary with whatever is the letter of the CD drive being used)

```
D:\>runme.bat
```

If running ENCAL on a system other than windows, then ENCAL should be run using

```
D:\>java gui/Encal
```

Appendix G

Mr Furnass helpfully provided a detailed set of questions covering the nuts and bolts of the new implementation. They are shown with our answers below.

The setup screen and related features

layout and usability is acceptable Yes: very good.

radio buttons and check boxes actually change the way ENCAL works The 'shelves first' button certainly works; couldn't evaluate the BODMAS button.

Main screen and general features

The general layout is acceptable; a particular panel can be sucessfully hidden Yes (congratulations, that can't have been easy).

Expression propagation occurs when expected and the smiley faces are updated accordingly Yes, works well.

The speed at which the software runs is acceptable The speed is much better than the Toolbook version

The reliability of the system in terms of crashes (more specifically, memory leaks) is acceptable Couldn't break it

The calculator representation

The presentation is acceptable; the full infix expression can be hidden (using the setup screen); the look inside panel can be hidden Yes, exactly as specified

All keypad buttons work as expected Yes

The infix expression is displayed correctly Yes

The correct result is displayed when '=' is pressed and the current calculator expression is well formed Yes

Both evaluation methods (left-to-right and BODMAS) are implemented correctly Yes

A suitable error message appears when the divide by zero anomaly is encountered This is an extension on the original ENCAL which did not implement division. The error message is well-phrased for its intended users.

The look inside panel DOES NOT display any sub-expressions This is a limitation.

Expressions are correctly propagated from other representations to the calculator Yes - and fast, too.

Expressions correctly propagate from the calculator to the core Again yes.

The iconic representation

A new shelf can be dragged into an empty shelf slot and added to the representation Yes

A new book can be dragged, but NOT ADDED to the representation Yes - still a limitation

A particular shelf or book CANNOT be deleted Correct

A particular book CANNOT be moved to another shelf yes

The ONLY expressions that can be propagated to the representation are in the form x**,** $x+y$ **or** $x \times y$ Yes

As the GUI does not allow books to be added to shelves, the number of expressions that can be propagated to the core is very limited Unfortunately yes.

Selecting shelves first or books first works with the limited set of expressions that can be propagated to the representation Yes

Error messages are displayed Yes

The data tree representation

The representation correctly updates itself from the core when necessary Yes

The node spacing and positioning is acceptable Precisely as specified. However, the new implementation omits the rectangles to be drawn in the tree, corresponding to parentheses in the calculator representation. (Our own explorations suggest that adding the rectangles will be quite easy.)

The arrows have no heads and for some strange reason, are all slightly offset from where they should appear onscreen yes, is that a Java bug?

No functionality to allow direct user interaction with the data tree has been implemented Correct. But as our future aims include entirely replacing the existing ENCAL design for direct tree editing, that's all to the good.

Error messages are displayed when the Undo and Arrange buttons are pressed Yes, that's good practice